‘*DevOps*’ - Path To Success

QAI

Logistics

- Class times
- Breaks
- Lunch
- Restrooms
- Other

QAI

Introduction

Name

Your Role

Your Experience with DevOps & Tools

5 = Expert, 4 = Very Experienced, 3 = Experienced, 2 = Some Experience,

1 = Very little/No experience

Why are you here

Anything else we should know

QAI

Day 01 – The Big Picture

Demystifying DevOps

- **What is DevOps?**
- **Why DevOps?**

DevOps Tools Introduction and Live demo

- **What is TDD?**
- **Demo Using TDD?**
- **BDD**
- **Selenium**
- **Emergent Design**
- **Configuring Static Code Analysis**
- **Configuring Mutation Analysis**
- **Jmeter (Performance Testing Tool)**
- **ZAP (Penetration Testing Tool)**
- **Static code Analysis - Sonarqube**
  - Sonarqube Rules
  - Sonarqube Dashboard
  - Sonarqube Reports
  - SQALE Rating

QAI

Day 02 – The Big Picture

DevOps Tools Introduction and Live demo

- **SCM : Live Demo** (Hands on Lab by students)
  - Understanding Basics of Git
  - Github – Create an account and fork your application code
  - Git clone the github code , Use maven to compile and package java source code
  - Branching Strategy
  - CI/CD Principles

DevOps Tools Practical

- **Defining and Implementing Commit Stage Pipeline**
  - Jenkins Configuration
  - Jenkins Pipeline Concept
  - Configuring Testing Pipeline/Reports View (On Jenkins)
  - Configuring Static Code Analysis Configuration (On Jenkins)
  - Configuring Code Coverage Analysis
  - Integration With SonarQube
- **Configuring SonarQube Build**
  - Configuring SonarQube rules For Java

QAI

Day 03/04 – DevOps Technical Essentials

Day 03 - Docker

- **Container Technology**
  - Understanding of Virtualization
  - Diving in to Docker
  - Configuring and Implementing Acceptance pipeline

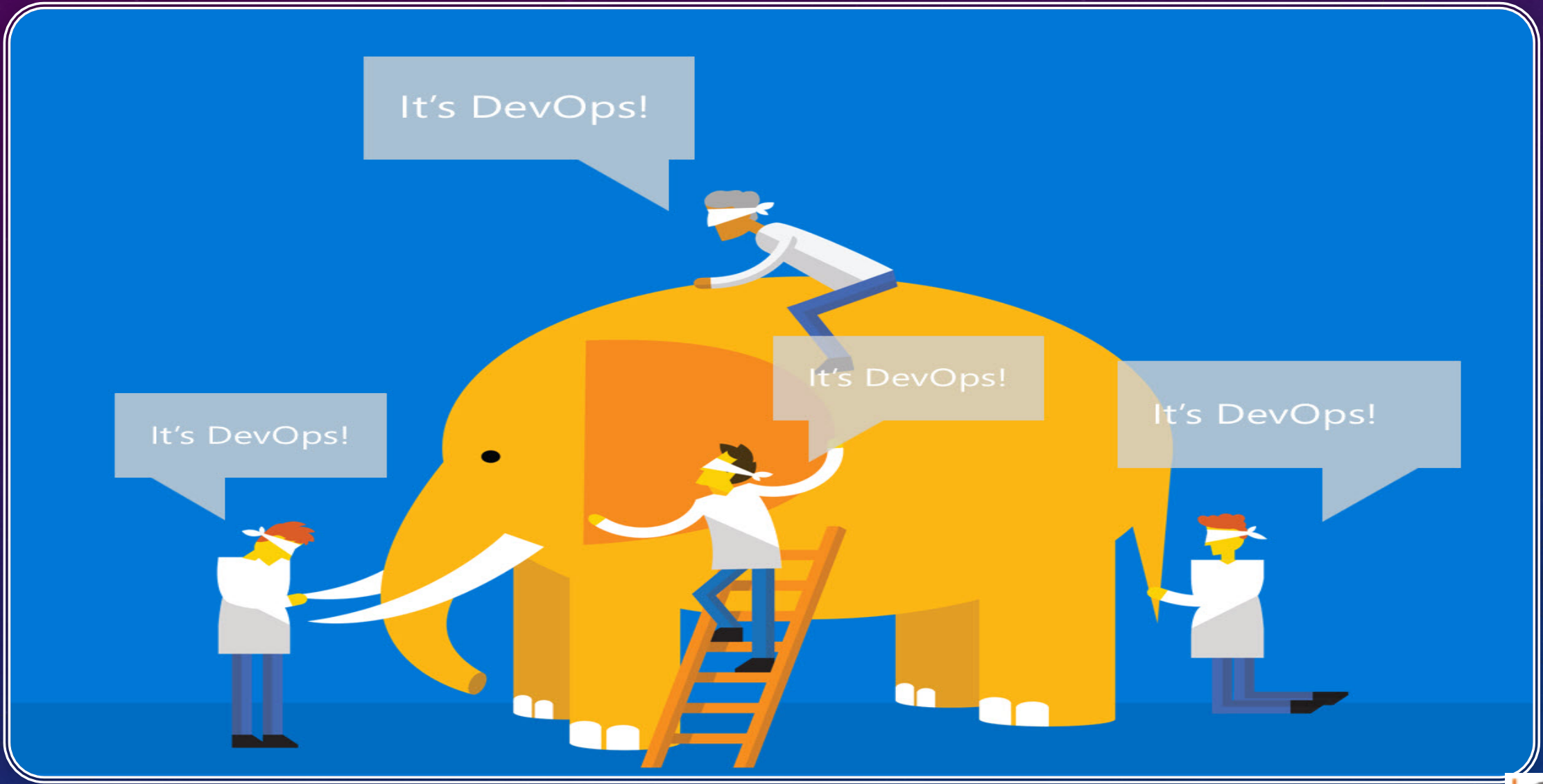
Day 04 - Ansible

- **Infrastructure as Code**
  - Ansible Deep Diving
  - Configuring and Implementing Acceptance pipeline

QAI

DEVOPS MINDMAP

What is DevOps ?



QAI

What is DevOps ?

Vooza

WHAT IS _____?

DevOps
Three Way
High Level Overview
Why?
Current Challenges
Models
What is DevOps?
Automation & Engineering Practices

Why DevOps ?

DevOps is in Top 3 CIO Initiatives

Gartner & State of DevOps Report 2017

"By 2020, half of the CIOs who have not transformed their teams' capabilities will be displaced from their organizations' digital leadership teams" – Gartner

2017 State of DevOps Report

Presented by:

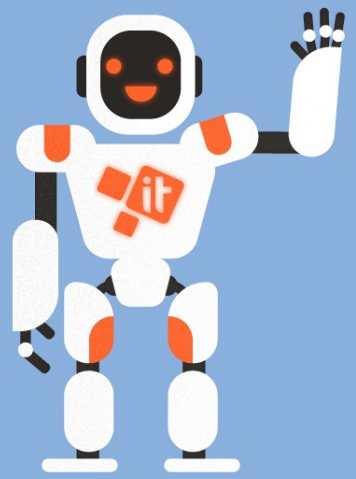
puppet + DORA

DEVOPS RESEARCH & ASSESSMENT

Adoption: DevOps Teams in Organizations

16% 2014 → 27% 2017

https://puppet.com/resources/whitepaper/2017-state-devops-report/
http://www.gartner.com/binaries/content/assets/events/keywords/infrastructure-operations-management/iome5/gartner-predicts-for-it-infrastructure-and-operations.pdf



46x
more frequent code deployment

440x
shorter time from commit to deploy

96x
faster TTR (time to recover)

5x
lower change failure rate

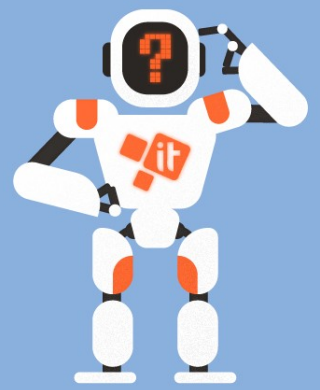
High-performing teams love automation.
They do less manual work and have automated:

33% more of their configuration management

30% more of their deployments

27% more of their change approval processes

27% more of their testing



QAI

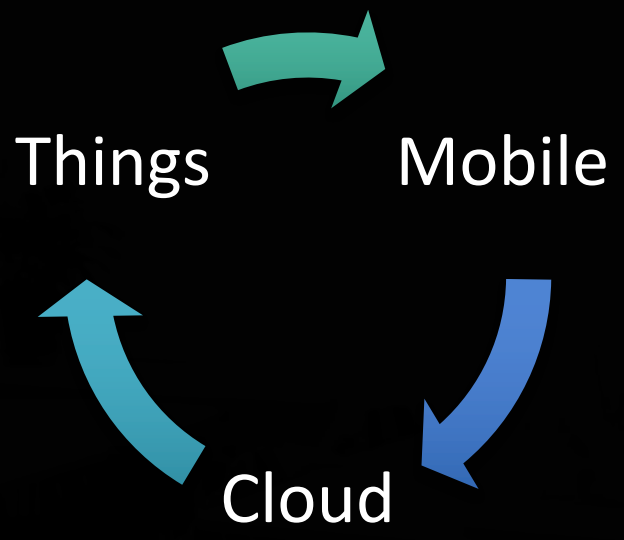
DEVOPS MINDMAP

SOFTWARE

"Software is eating the world."

"Every business is in the software business."

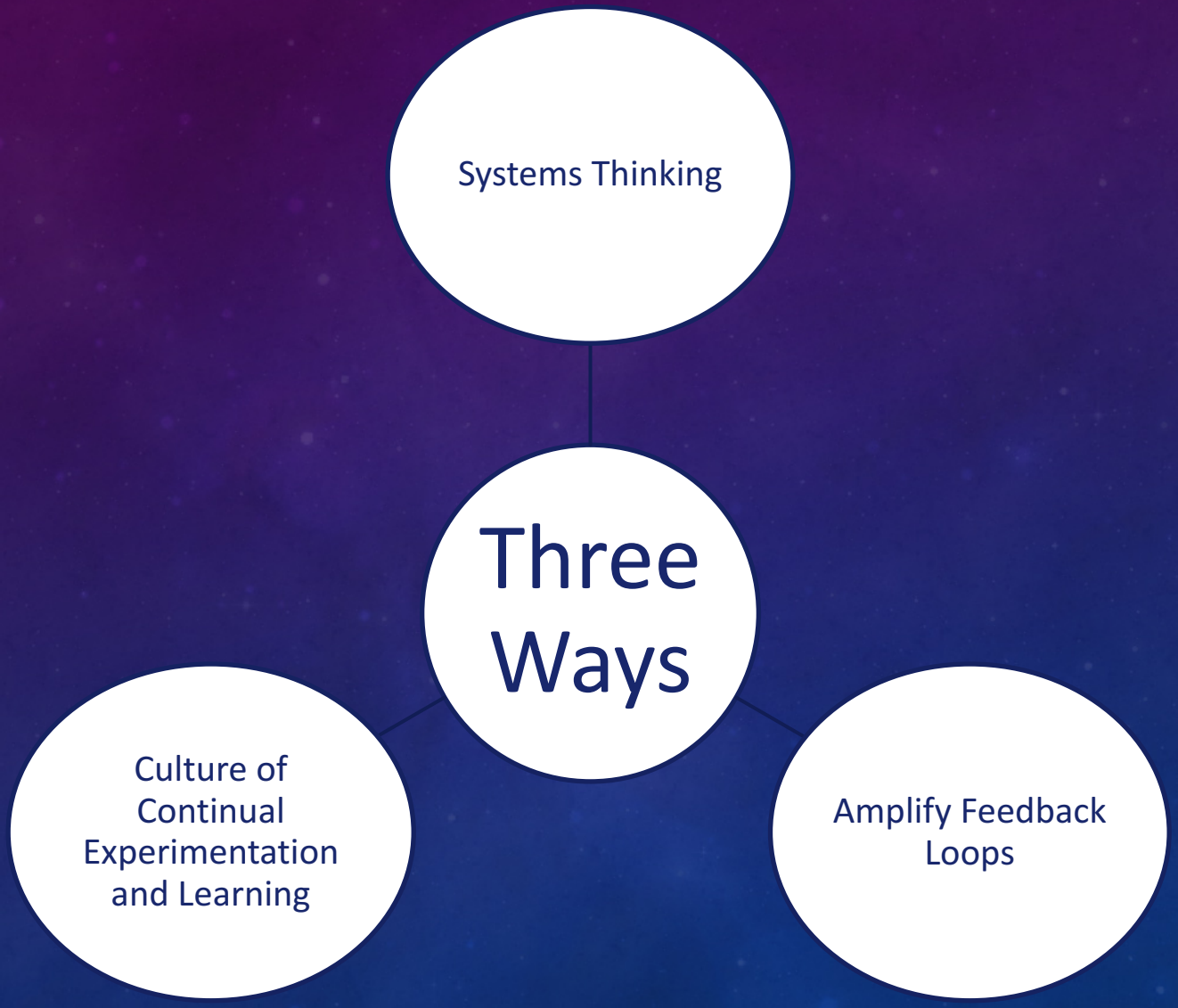
"It is an application economy."



HOW DOES SOFTWARE HELP YOU ACHIEVE YOUR GOALS?

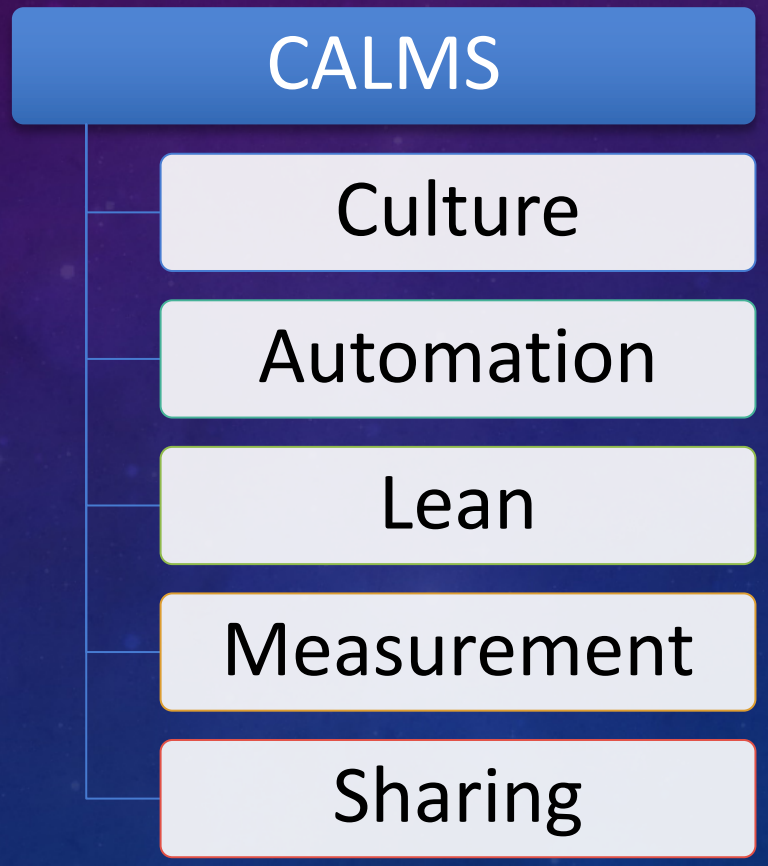
DevOps
Three Way
High Level Overview
Why?
Current Challenges
Models
What is DevOps?
Automation & Engineering Practices

THREE WAYS


Systems Thinking
Three Ways
Culture of Continual Experimentation and Learning
Amplify Feedback Loops

DevOps
Three Way
High Level Overview
Why?
Current Challenges
Models
What is DevOps?
Automation & Engineering Practices

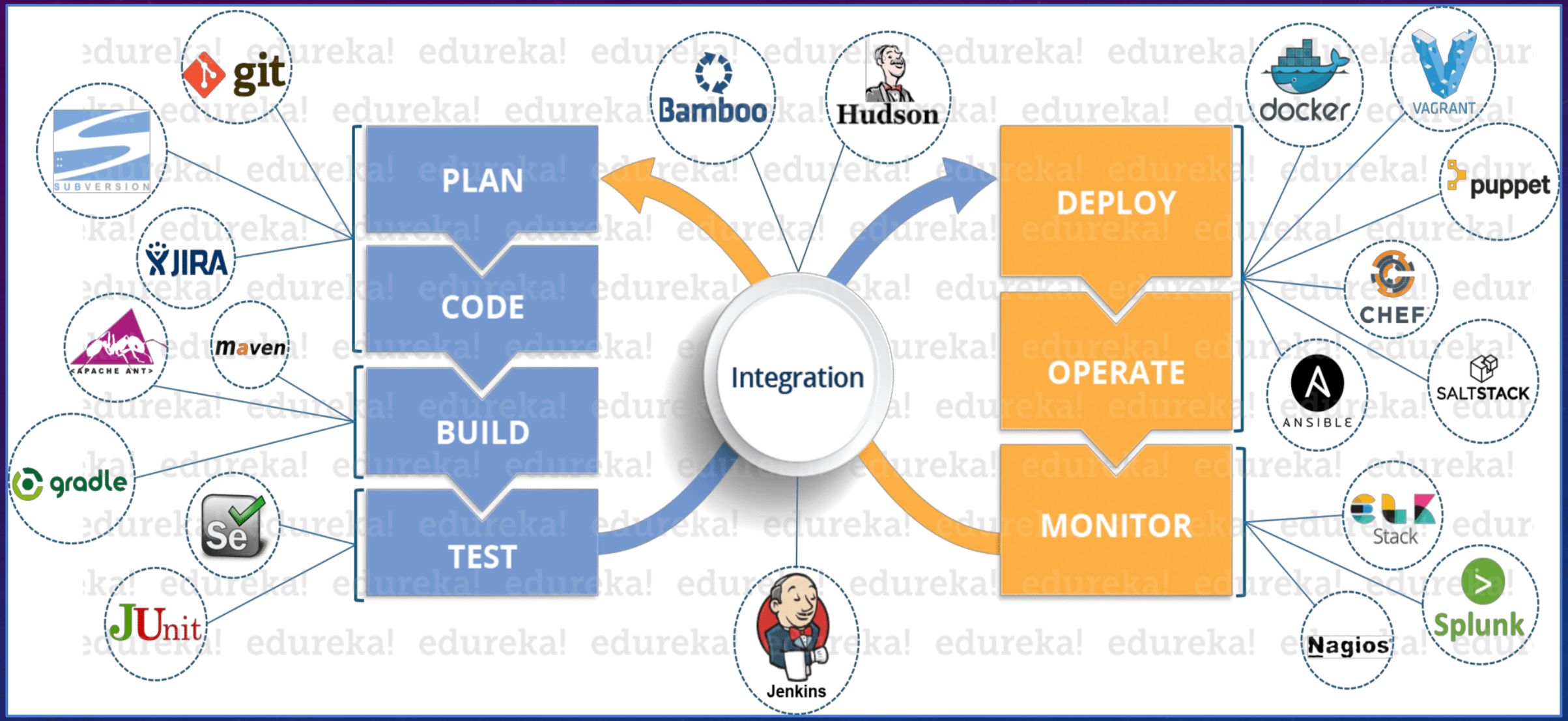
DEVOPS MODEL



DevOps
Three Way
High Level Overview
Why?
Current Challenges
Models
What is DevOps?
Automation & Engineering Practices

ENGINEERING PRACTICES

DevOps: Tools & Integration



QAI

TESTING

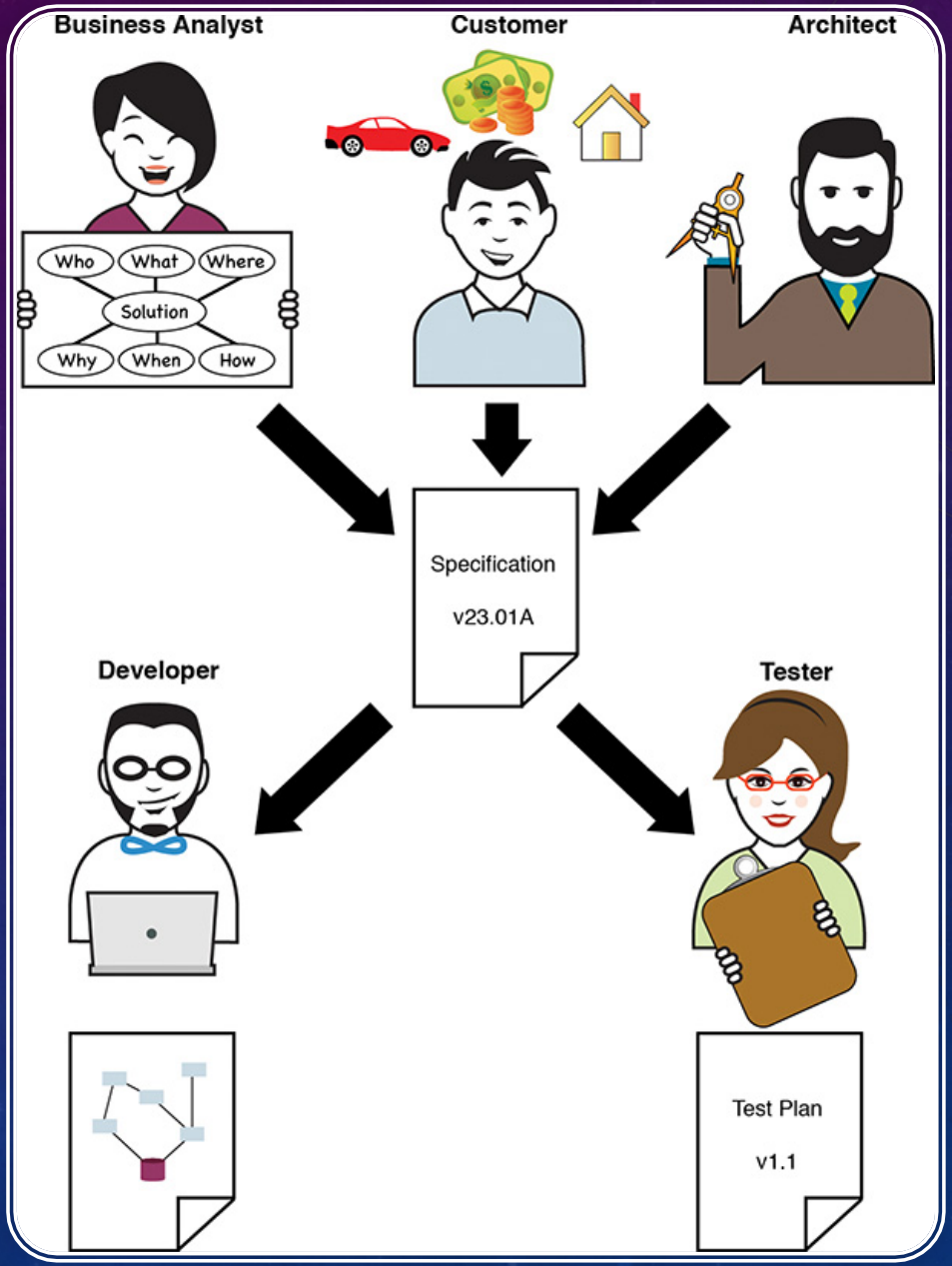
TESTING OBJECTIVES

- Critiquing
  - Test something that is finished and needs evaluating
  - Does it deviate from Specification?
  - Are there any defects in it?
- Supporting
  - Test is about ensuring the safety, sustainable pace, and the team's availability

Overall objective is to reduce the risk in software by inspecting it critically

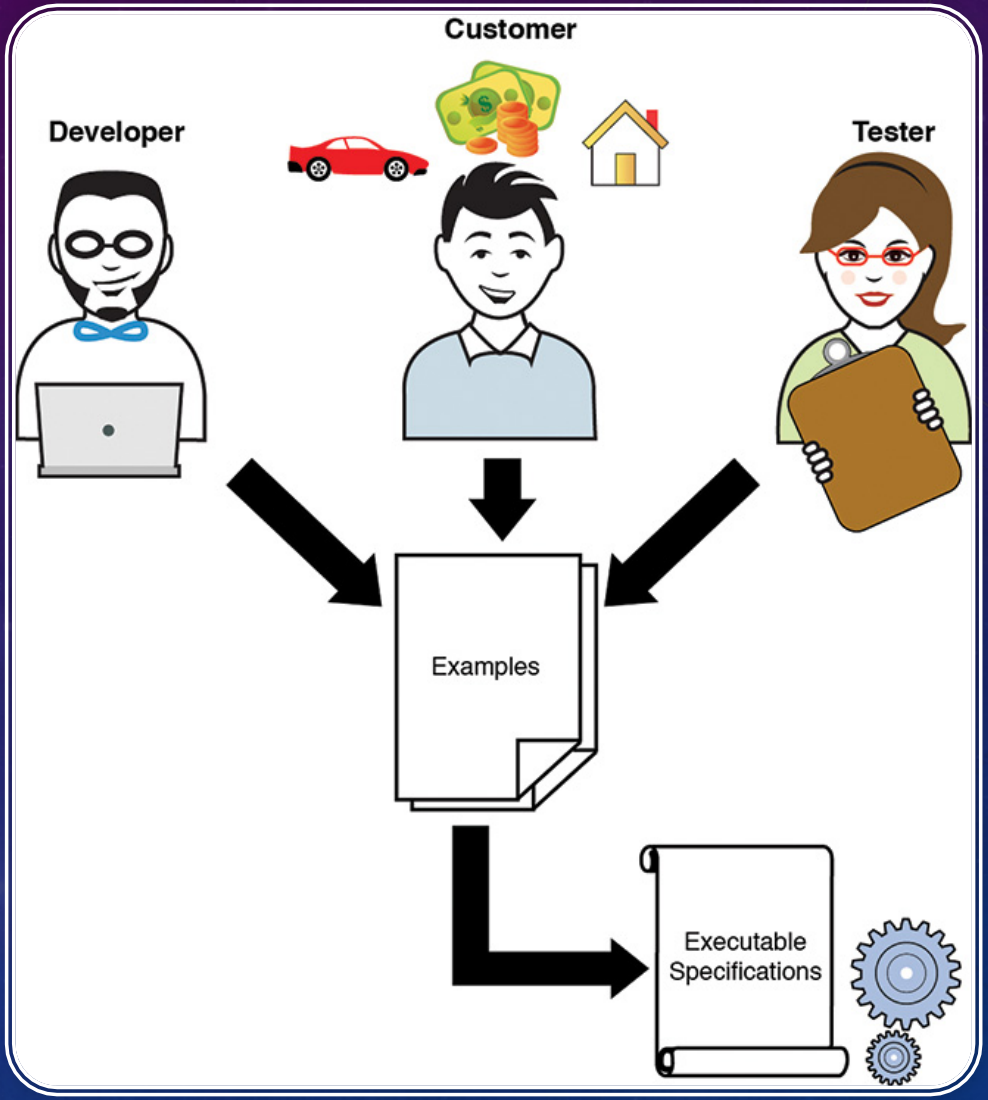
TESTING STYLES

- Traditional Testing
  - Process-oriented
  - Independent
  - Formal

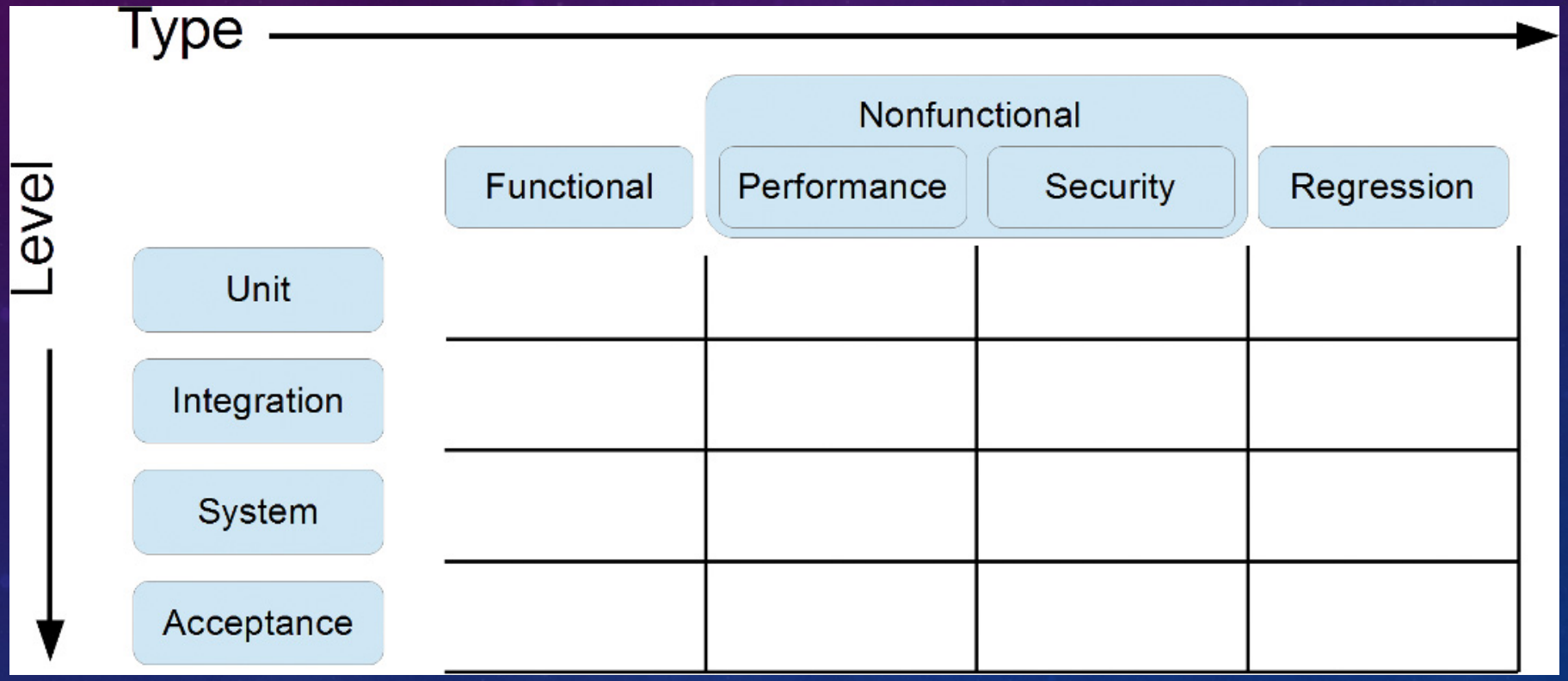


TESTING STYLES

- Agile Testing
  - Proactive
  - Integrated
  - Collaborative



TEST LEVEL AND TYPES



AGILE TESTING QUADRANTS

Business facing

Guide development

Examples A/B tests Story tests (written first) UX (user experience) testing Prototypes Simulations Q2	Exploratory testing Workflows System integration (business oriented) Usability testing User acceptance testing Q3
Q1 Unit tests Component tests (code level) Testing connectivity	Q4 Performance testing Load testing Security testing Quality attributes (… ilities)

Critique the product

Technology facing

TEST DRIVEN DEVELOPMENT

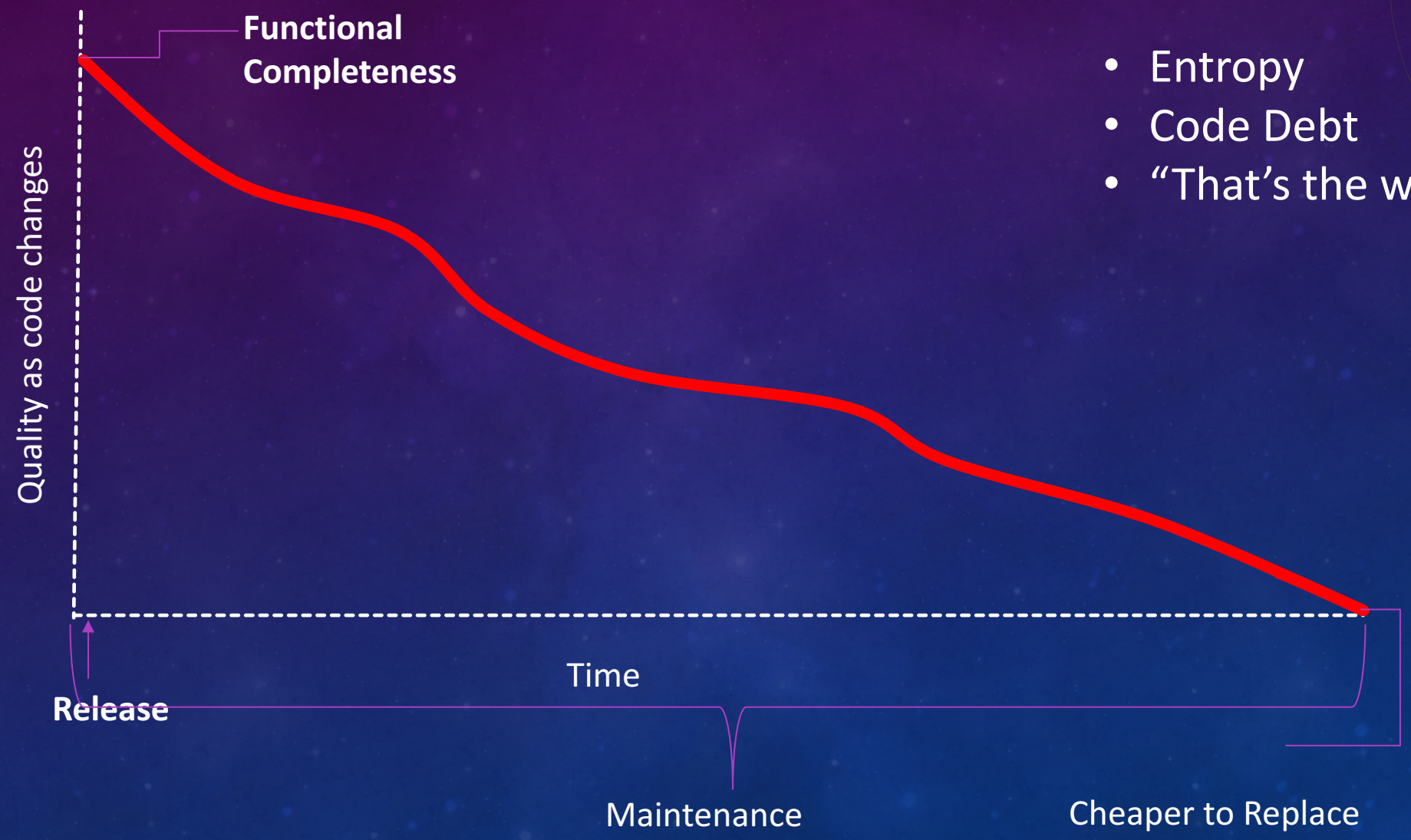
TDD DEMO

EMERGENT DESIGN

ANALOGY – EVOLVING SYSTEMS

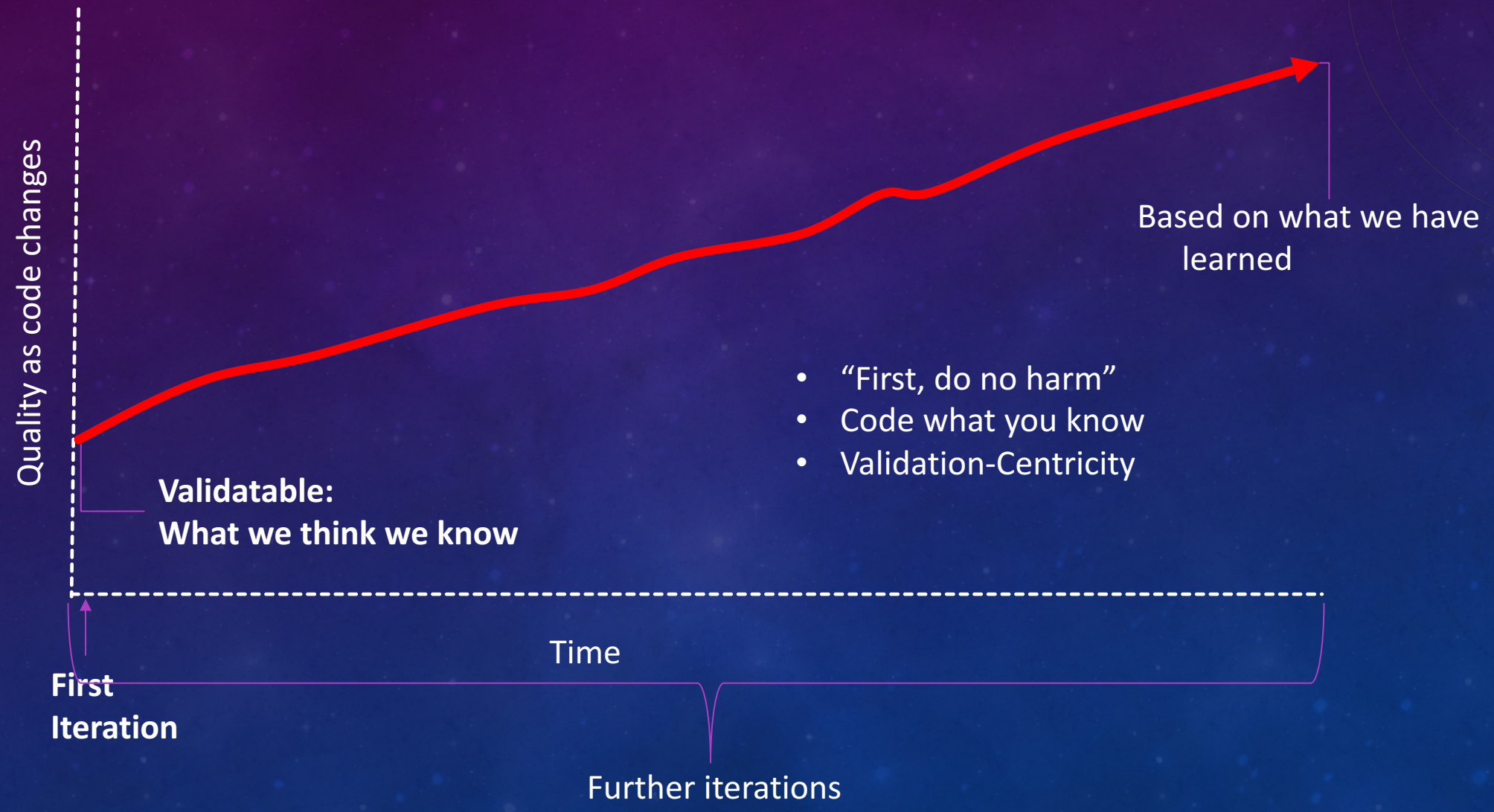
"All software systems decay over time to the point that, eventually, replacing them is less costly than maintaining them."

INEVITABLE DECAY



- Entropy
- Code Debt
- “That’s the way it is”

REJECTING DECAY: CHANGING CHANGE



- "First, do no harm"
- Code what you know
- Validation-Centricity

APPROACH

Phases Analysis, Design, Code and Testing are highly interrelated

Design

- Is Reflection of Domain

Coding

- Practicality of Design
- Teaches about Problem Domain

Testability Consideration

- Leads to Evaluation of Design

Analysis

- Questions arising during designing, coding and testing lead to better domain analysis

THE LAWS OF SOFTWARE EVOLUTION

A series of 8 laws formulated by Lehman & Belady starting in 1974, including:

- *The Law of Continuing Change*
- *The Law of Increasing Complexity*
- *The Law of Continuing Growth*
- *The Law of Declining Quality*

The laws can be summarized as follows:

- A system that is being used undergoes continuing change or degrades in effectiveness.
- A computer program that is changed, becomes less and less structured. The changes increase the entropy and complexity of the program.

Source: Wikipedia on "Lehman's Laws of Software Evolution"

WHAT IS DESIGN?

- Risk Mitigation
- Opportunity to reduce waste
- Documentation of Intent
- The thought process behind software
- Exists at multiple levels
  - Architecture
  - Design
  - Code

WHEN DO WE DO DESIGN?

- After Analysis?
- Before Implementation?
- Do we test our design?
- What if we get wrong?
- What if we overdo it?
- How much design is enough design?

EMERGENT DESIGN

It is the process of evolving systems in response to changing requirements, better understanding of existing requirements, and in response to new opportunities that arise from new technology, better ideas, and a changing world.

EMERGENT DESIGN

Emergent Design is a fancy name for the resulting design that "emerges" from the synergy of combining Refactoring together with TDD, Continuous Integration and Automated Testing.

"As a result, you'll find the balance of work changes.

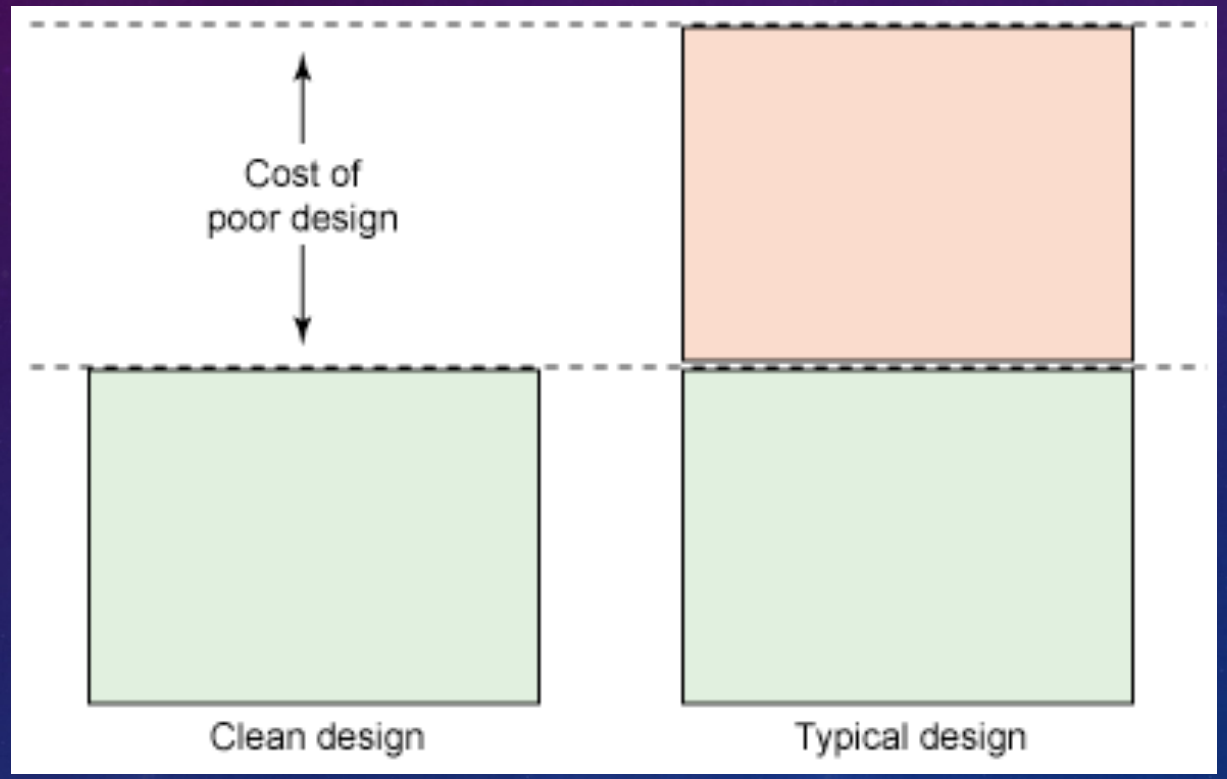
- Design, rather than occurring all up front, occurs continuously during development.
- The cumulative effect of these small changes can radically improve the design.
- You learn from building the system how to improve the design.
- The resulting interaction leads to software with a design that stays good as development continues.
- It is the exact reverse of the normal notion of software decay."

— *Martin Fowler, Refactoring: Improving the Design of Existing Code*

A lack/lapse of discipline in *any* of these technical practices will result in the accrual of *technical debt!*

TECHNICAL DEBT & INTEREST

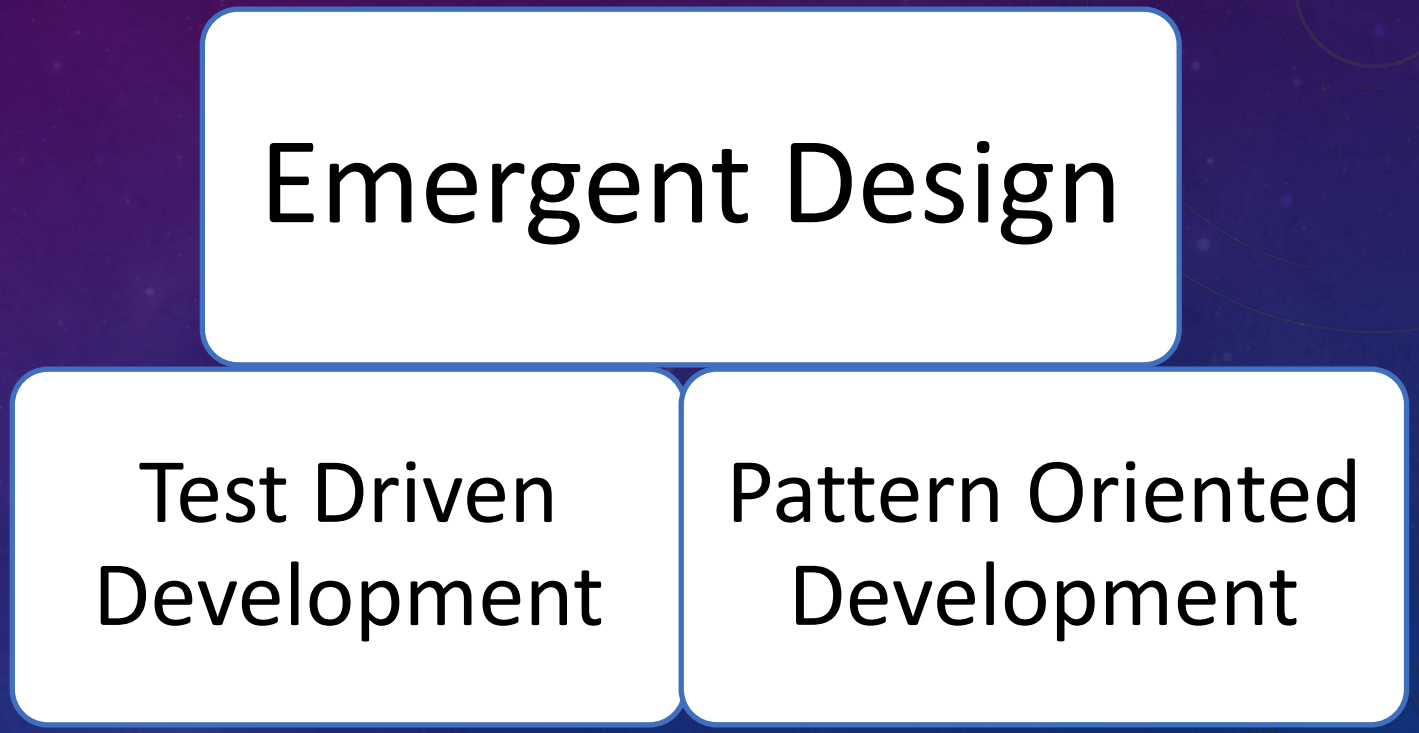
Difference in effort required to add a new feature in a cleanly designed system, versus a typical system that contains a lot of technical debt.



You can think of the inherent complexity as the principal, and the extra effort imposed by previous expedient shortcuts as the interest.

EMERGENT DESIGN

Emergent design depends on the disciplines of TDD and POD



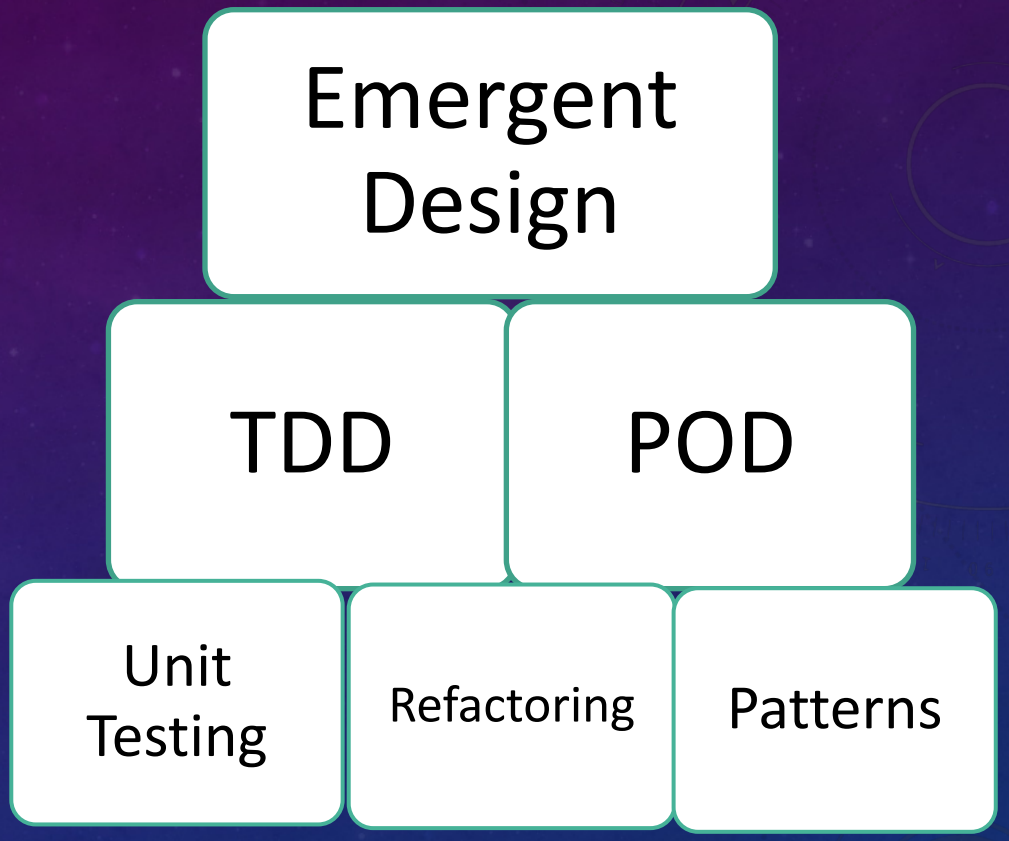
TDD & POD

TDD

Provides coverage for refactoring

Drives toward high quality design

•



TDD and POD depends upon knowledge and skills like

- Unit Testing
- Refactoring
- Patterns

QUALITIES

Follow the Design Quality Attributes to achieve higher Maintainability of Code

Encapsulation	• Encapsulate behavior & state
Cohesion	• Defines singleness of purpose
Coupling	
Redundancy	
Readability	
Testability	

AVOIDING OVER DESIGN

- Complexity When You need it
  - But not Before you need It
- Good practices reduces risk
- Risk is wasteful
- Over-Design is wasteful

- Emergent Design uses
  - Refactoring to reduce over-design and enable low-risk change
- Principle can guide us…
- Practices can protect us...
- Discipline can empower us...
- You cannot Stop thinking!

WHAT ARE THE CHARACTERISTICS OF THE EASILY MAINTAINABLE CODE?

- Code is readable
  - can see what the individual things do
  - can see how individual things interact
- Can change code in one place without it adversely changing behavior in another place
- When need to make a change, only need to change it in one place
- In other words: readable, independent, efficient and of course correct

TESTABILITY AND DESIGN

- Considering how to test your objects before designing them is, **in fact, a kind of design**
- It forces you to look at:
  - the **public** method definitions
  - what the **responsibilities** of the objects are
- Easy testability is tightly correlated to loose coupling & strong cohesion

TESTABILITY

- Code that is difficult to unit test is often:
  - Tightly coupled: “I cannot test this without instantiating large number of objects”
  - Weakly cohesive: “This class does so much, the test will be enormous and complex”
  - Redundant: “I’ll have to test this in multiple places to ensure it works everywhere”

UNIT TESTING

- Tests at a low, or granular level
- Each test confirms that the code accurately reflects one intention of the system
- Test per class is often a natural fit
- A good test of our thought process:
  - If we've designed classes to do one thing (strong cohesion), then we should be able to test their functionality
  - If classes do not create side effects in other classes, (loose coupling), then we should be able to test them individually

TDD: BENEFITS

- My personal experience using TDD for many years had been
  - Keep defect counts low
  - Refactor without fear
  - Produce simpler, better code
  - Program without stress

TDD & EMERGENT DESIGN DEMO

CASE STUDY 1

EMERGENT DESIGN

REQUIREMENTS

- System is required to ***monitor*** two hardware like **Chips** and **Cards**.
- System should provide an interface to request the status of both of these types of hardware
- System should support ***multiple Transmission*** mechanism for sending message
  - Status Message can be sent over either a **TCP/IP** connection or via **E-mail (SMTP)**
- System should ***multiple message Encryption*** mechanism.
  - Messages may be optionally encrypted with either **PGP64** bit encryption or **PGP128** bit encryption
- When sending status out for a chip, we want to queue the information to send it out no more than every 10 minutes unless there is an error. Cards, on the other hand, send immediately
- A **configuration file** will contain information about which transmission method to use

FINDING ENTITIES AND BEHAVIORS

- This has us focus on the “things” we have, first. Then, we handle differing behaviors through the use of polymorphism
- If more than one behavior varies, can result in tall class hierarchies...
  - which are hard to test
  - which have unintended coupling
  - which have redundancies
  - which promote weak cohesion

ACCOMMODATING CHANGE WITH SPECIALIZATION

- Let’s walk through a potential way our problem could evolve. We start with Chip and TCP/IP and add function one step at a time. We accommodate this through specialization
- The result is not pretty (except as in pretty common)

REQUIREMENTS AS USER STORIES

- **Story 1**: Request the status of a chip, encrypt it with PGP64 bit encryption and send that status out via TCP/IP
- **Story 2**: Allow for not encrypting the status or using either PGP64 bit or PGP128 bit encryption. A configuration file will determine what (if any) encryption is needed
- **Story 3**: Support transmission via an e-mail connection. A configuration file will determine which type of transmission to use
- **Story 4**: Support getting and sending the status for a card as well

EMERGENT DESIGN: STARTING WITH STORY 1

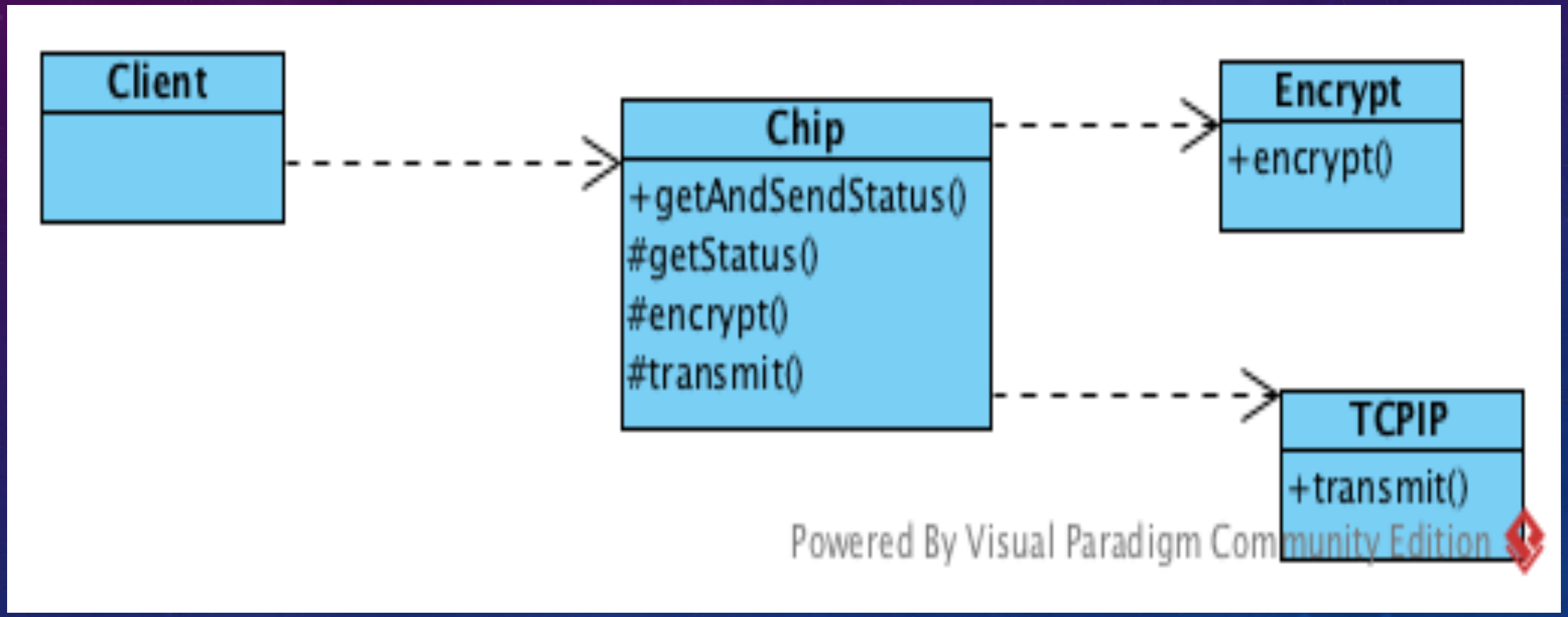
- System should provide an interface to request the status of a chip.
- Status of chip should be encrypted with PGP64 bit encryption
- Encrypted Status should be send out via TCP/IP
- We will follow our rules attempting to implement the simplest solution possible. This really means:
  - No extra function
  - Design for full system will emerge via refactoring

Follow JEDI principle

TDD STEPS

1. Write a test that expresses an intent of a class in the system
2. Stub out the class (enough to allow the test to compile
3. Fail the test (don't skip this)
4. Change the class just enough to pass the test
5. Pass the test
6. Examine the class for coupling, cohesion, redundancy, and clarity problems. Refactor.
7. Pass the test
8. Return to 1 until all intentions are expressed

SOLUTION DIAGRAMMED



THINKING FROM A TESTABILITY PERSPECTIVE

- What do we need to test?
  - getting status
  - encrypting a string
  - sending an encrypted string
- Writing the tests for the individual methods should be straightforward enough.
- Where should the methods lie?
- What's the easiest way?

Think But Don't Assume You will get the right answer in the beginning

SOLUTION DIAGRAMMED

```java
public class Encrypt {

    public String encrypt(String status){
        return "Encrypted Status";
    }
}
```

```java
public class Chip {
    public String getStatus(){
        return "ChipStatus";
    }
}
```

```java
public class TCPIP {
    public void transmitStatus(String status){
        System.out.println("Transmitted status");
    }
}
```

```java
public class Client {

}
```

WHY IS IT BETTER?

- It is clear what the pieces do and how they relate
  - Increases extensibility
  - Eases maintainability
- Sometimes requirements mislead us

STORY 2 : MULTIPLE ENCRYPTIONS

- Allow for using
  - **No Encryption**
  - **PGP64 bit Encryption**
  - **PGP128 bit Encryption**

REFACTORING SAYS TO RESTRUCTURE BEFORE ADDING

- Refactoring tell us we should change our code before adding new function
- That is, we keep the function same, but restructure our code to improve it
- This has the following advantages:
  - If we have been doing up front testing, our tests don't need to change
  - We always have something that works
  - If something breaks, we are more likely to know what caused it(one step at a time)

HOW CAN WE MOST EASILY TEST THINGS?

- Need to test every encryption
- Want to deal with using encryptions the same way
- Easiest way is if we can have common interface to encryptions

If interfaces aren't the same, then we have two choices:

Have Chip know differences and deal with them

Use lessons from Design pattern and hide the variations

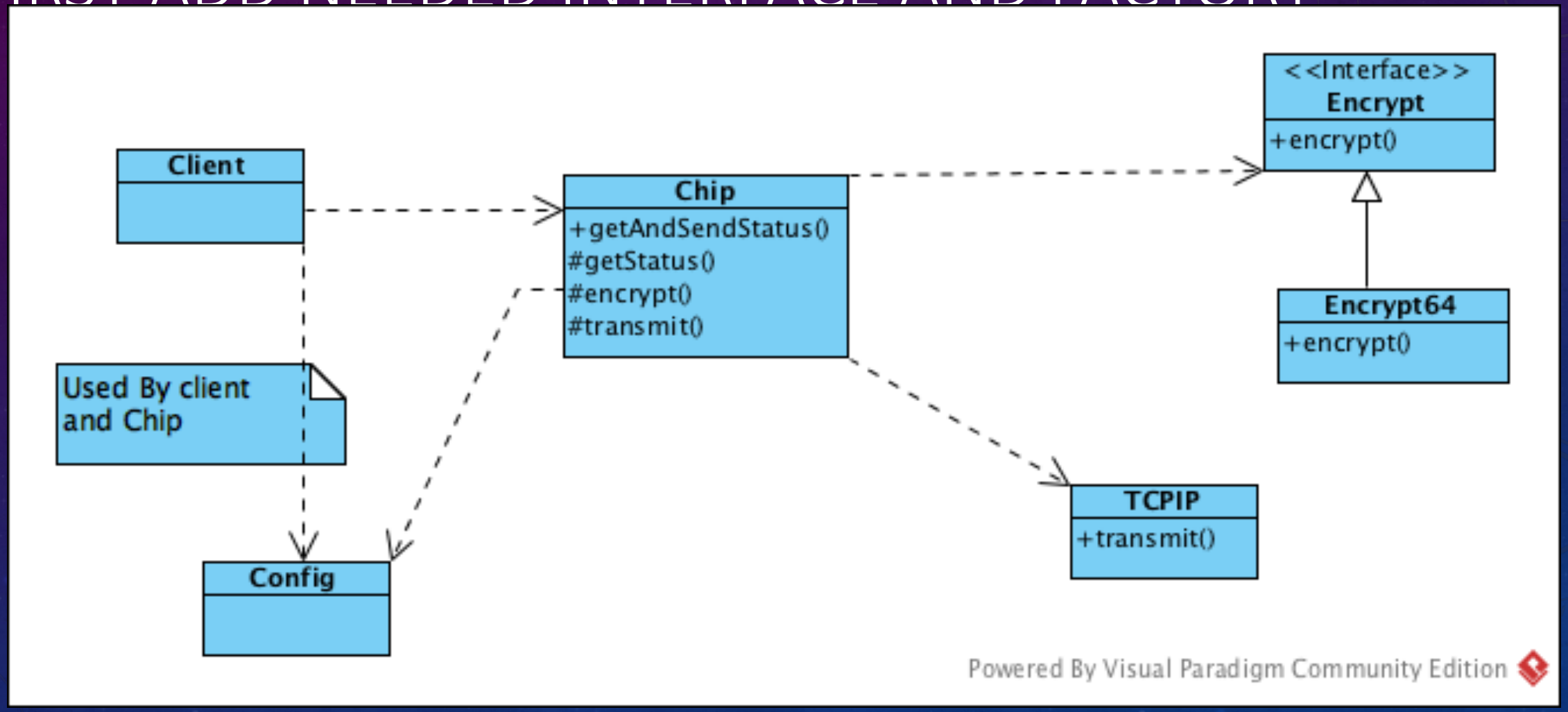
OPEN CLOSED PRINCIPLE

- Ivar Jacobson said:
  - ***"All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version"***
- Bertrand Meyer summarized this as:
  - *Software entities(classes, modules, functions, etc.)should be open for extension, but closed for modification*
- In Simple Term this means:
  - design modules so that they never change.
  - When requirements change, add new modules to handle things

STORY 2 : REFACTORING TO OCP

1. First refactor code so that new function can be added following OCP
2. Then add new code

FIRST ADD NEEDED INTERFACE AND FACTORY

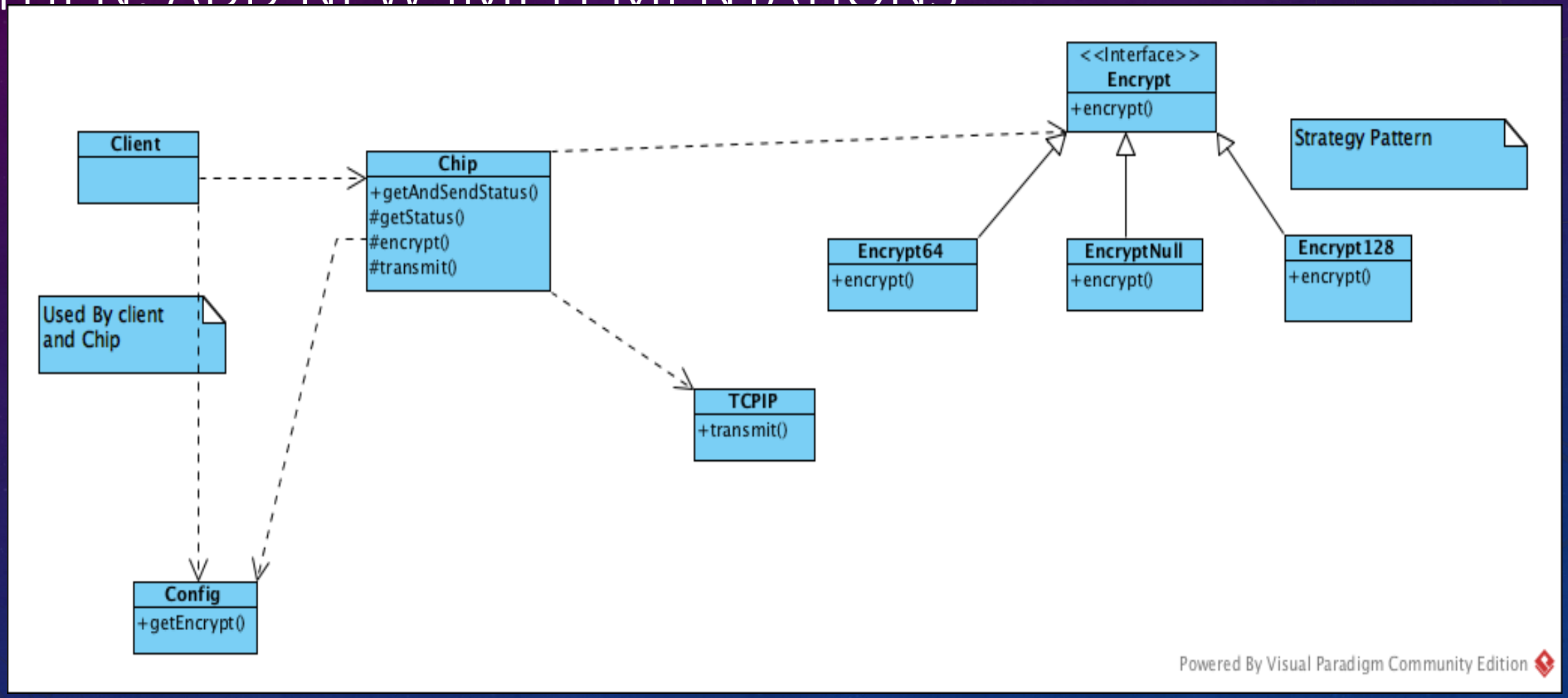


NOW, WE CAN PUT IN THE NEW FUNCTION

- To put in new function
  - someone must determine which encryption type to use
  - Let's say we have a configuration object that can do this for us
- Client object can ask configuration object which to use
- Chip object can then be told what behavior is needed
- Now add encrypt128 and no encrypt options

Note: *Chip object could be responsible for talking with configuration object, but then Client must give Chip all the information the configuration object needs*

THEN, ADD NEW IMPLEMENTATIONS



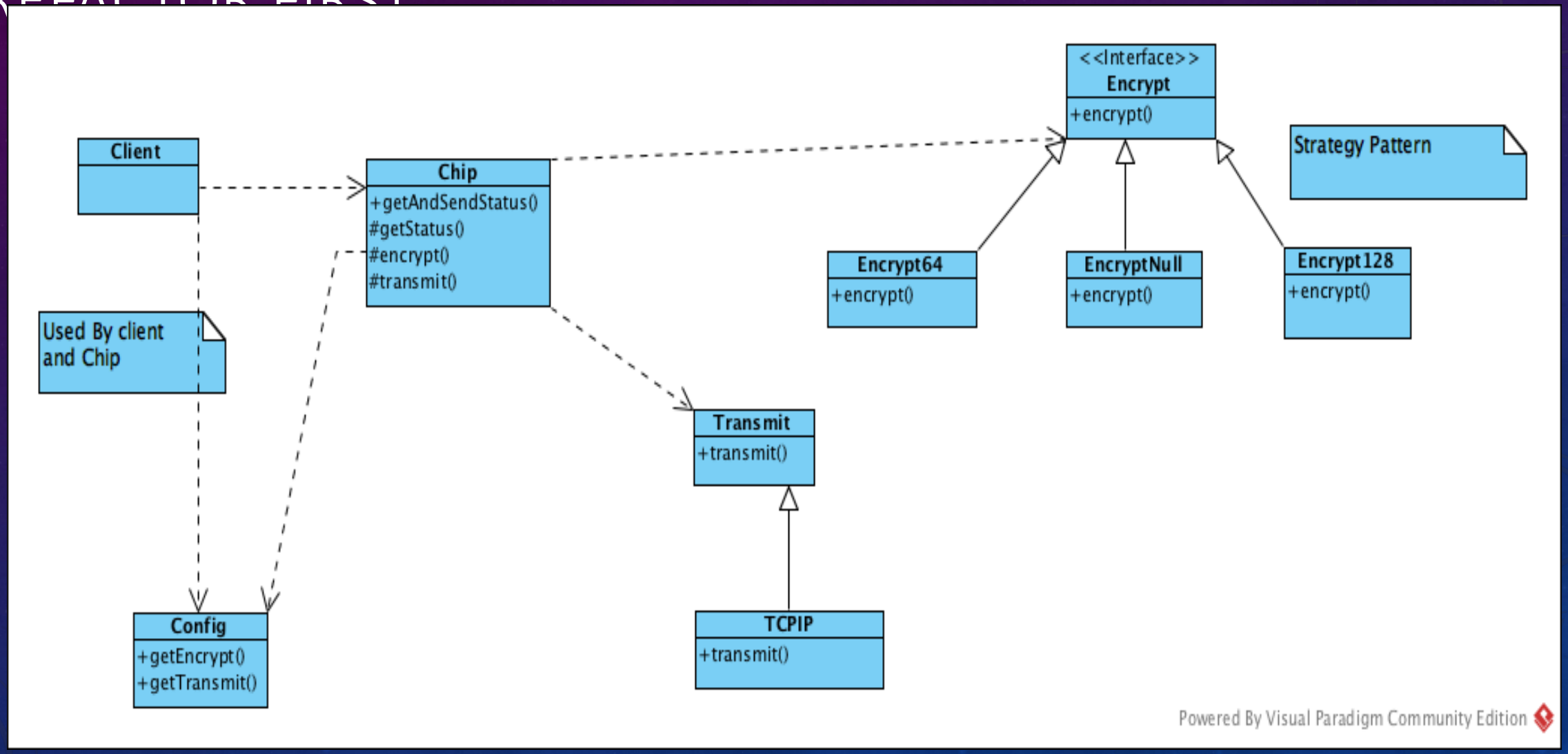
STORY 3

- Support transmission **via an e-mail connection**. A configuration file will determine **which type of transmission** to use
- Testability tells us to deal with concept of Transmission (don’t have to test all combinations)
- Design Patterns tell us same thing

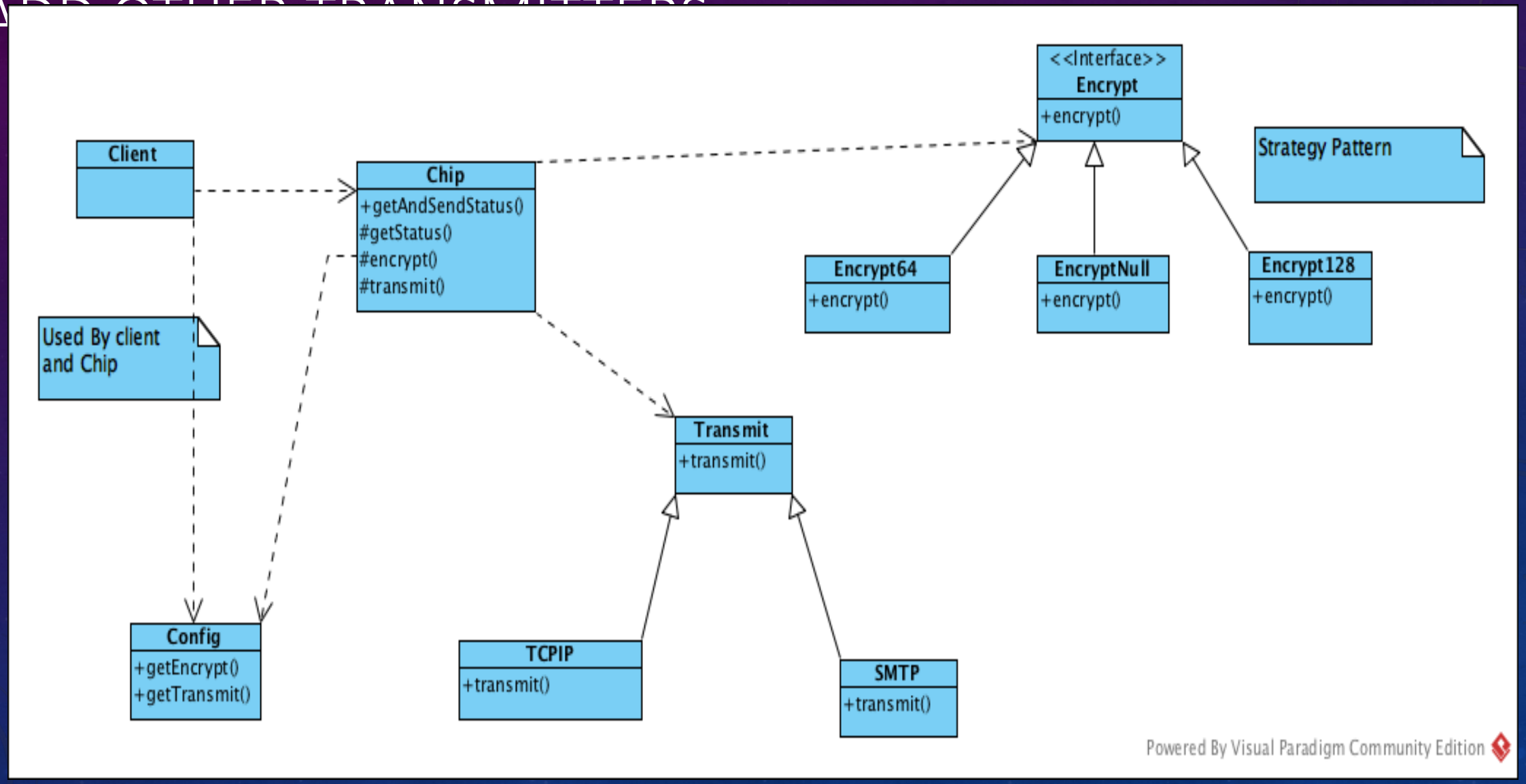
WANT TO FOLLOW GANG OF FOUR ADVICE

- *Because transmission is going to vary*, we want to encapsulate it and contain it in the using class (the Chip)
- To implement this, ***first pull out the existing transmission functionality into its own class***
- Want to Follow Gang of Four Advice
- Note that **there is no extra cost** caused by us doing this now instead of when we first noticed it was possible

REFACTOR FIRST



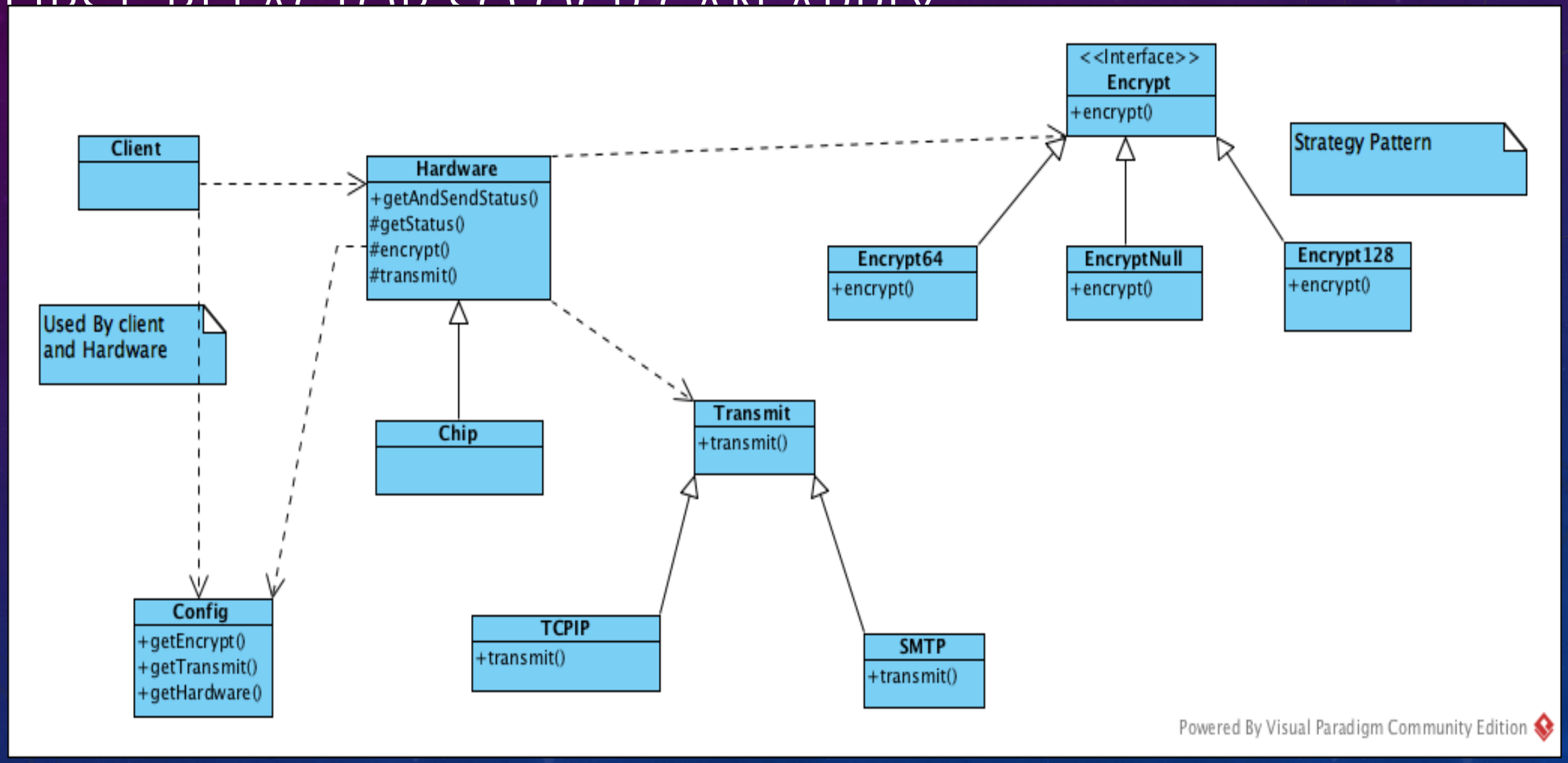
ADD OTHER TRANSMITTERS



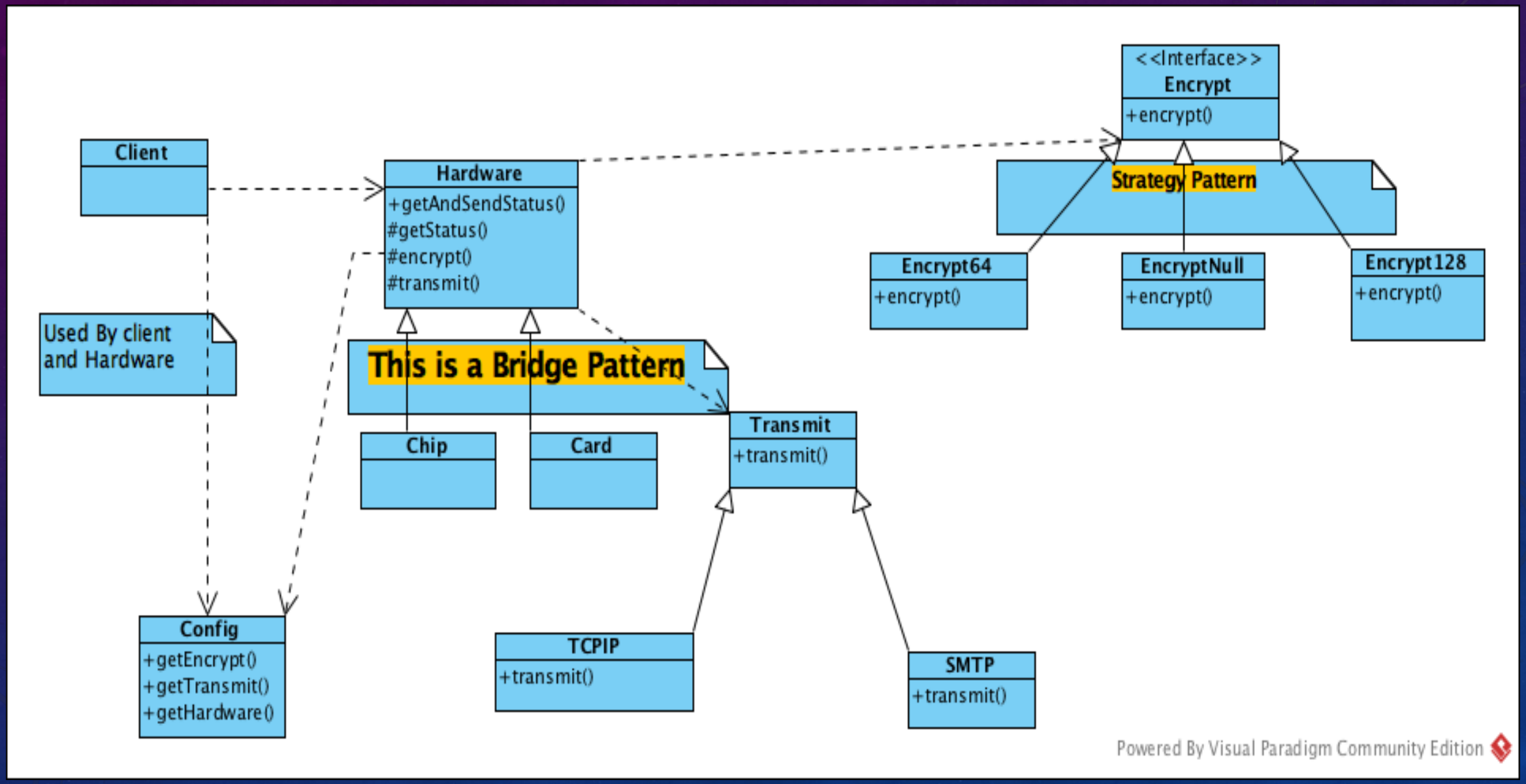
STORY 4 : SUPPORT CARDS

- For testing reasons, we don’t want to have to handle different functional test cases when we have Cards or Chips – we want to handle them together
- We want Cards and Chips to appear to be the same to the Client
- Story 4: Support Cards
- We will implement the code with our two-step Refactor-OCP shuffle

FIRST, REFACTOR SO OCP CAN APPLY



THEN, ADD NEW FUNCTION



DEVELOPER PRACTICES

Follow Kent Beck's Four Rules of Simple Design

1. Runs all the tests

DEVELOPER PRACTICES: ALLOW EMERGENCE TO OCCUR

SUMMARY OF EMERGENT DESIGN PRACTICES

MUTATION TESTING

WHY WE NEED MUTATION TESTING?

Unit Testing shows the presence and not the absence of bugs

So even if we “Write all the test”, will this ensure our code actually works

This bring us to the set of questions like:

1. Have you written enough tests?
2. Have you covered every line, every branch, every path?
3. If semantic change is made to the code, will our test fail?

Test Sufficiency

Coverage % ??

Semantic stability ??

Coverage does not prove that you have tested every line and branch

Mutation Testing aims to discover the semantic inconsistencies with respect to our test suite

ISSUE WITH COVERAGE METRICS?

"Code Coverage is a measure used to describe the degree to which the source code of a program is tested"

--Wikipedia

- Code Coverage is a metric
- Any metric can be easily manipulated
  - On purpose
  - Or by accident
- Code coverage lulls us to a false sense of security
- Code coverage cannot ensure test quality on its own only

MUTATION TESTING: DO'S AND DON'T'S

- Mutation Testing Only
  - Checks the relevance of your Unit Tests
  - Points out potential bugs in your tests as well as code

- Mutation Testing does not cover
  - Validating the application functionality
  - Checking the performance of system
  - Any kind of end to end testing

MUTATION TESTING FRAMEWORK

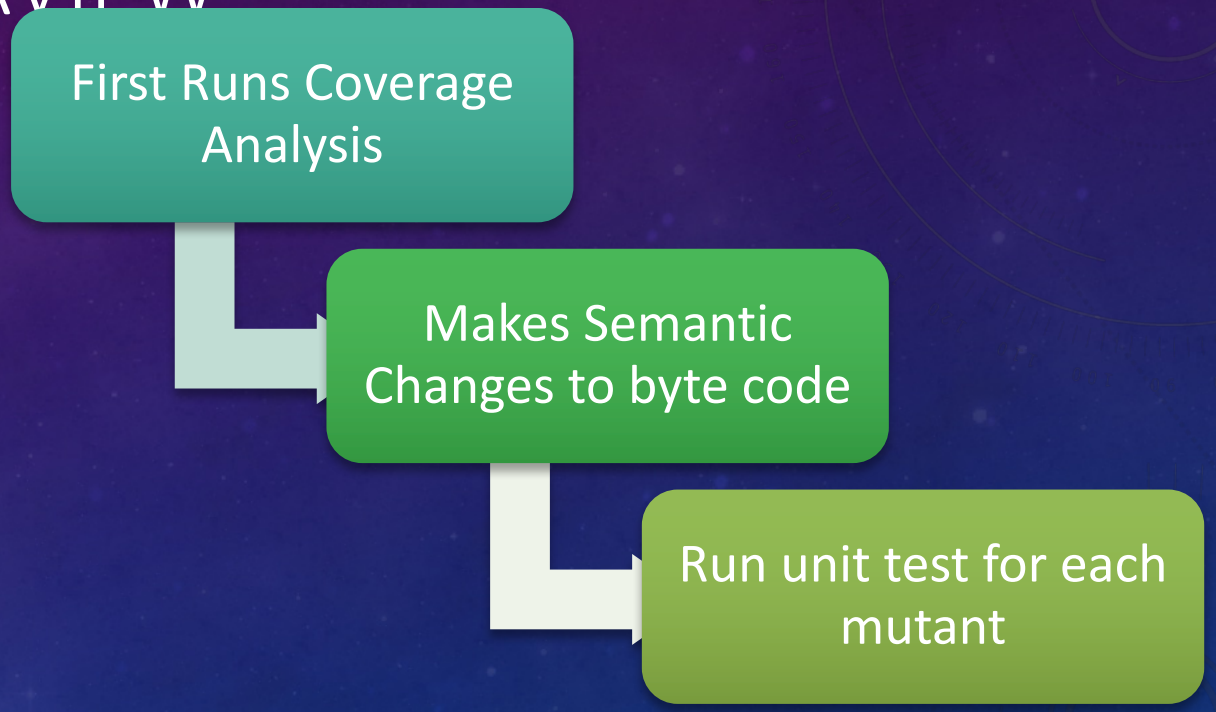
Framework available are like

- PITEST (www.pittest.org)
- Jtester (www.jtester.sourceforge.net)
- Jumble
- Grunt Mutation Testing (Javascript)
- Mutandis (Javascript)

PITTEST FRAMEWORK : OVERVIEW

Semantic Changes are called Mutants

If test fails for a mutant, then it is good test and if not then test is not a good one



Green mutant indicates:

- Tests that have been ignored or commented out
- Tests where asserts have been removed or not provided
- Laxity in TDD discipline, where some code have been added without proper test

PITTEST: AVAILABLE MUTATORS

Enabled Mutators

- Conditionals Boundary Mutator
- Increments Mutator
- Invert Negatives Mutator
- Math Mutator
- Negate Conditionals Mutator
- Return Values Mutator
- Void Method Calls Mutator

Disabled Mutators

- Constructor Calls Mutator
- Inline Constant Mutator
- Non Void Method Calls Mutator
- Remove Conditionals Mutator
- Experimental Member Variable Mutator
- Experimental Switch Mutator

MUTATOR: CONDITIONAL BOUNDARY MUTATOR

- Replaces the relational operators <, >, <=, >= with their boundary counterpart

Original Conditional	Mutated Conditional
<	<=
>	>=
<=	<
>=	>

```
if(a<b){
        // process something
}
```

mutated to

```
if(a<=b){
        // process something
}
```

MUTATOR: NEGATE CONDITIONAL MUTATOR

- Replaces the conditional operators with their negated form

Original Conditional	Mutated Conditional
<	>
>	<
<=	>
>=	<
==	!=
!=	==

```
if(a==b){
        // process something
}
```

mutated to

```
if(a!=b){
        // process something
}
```

MUTATOR: REMOVE CONDITIONALS MUTATOR

- Removes all conditional statements such that the guarded statements always execute

```
if(a==b){
        // process something
}
```

mutated to

```
if(true){
        // process something
}
```

Specialized version of mutator ensure that block never executes

```
if(a==b){
        // process something
}
```

mutated to

```
if(false){
        // process something
}
```

If else block exist then else block will execute

```
if(a==b){
        // process something
}else{
        // process something
}
```

mutated to

```
if(false){
        // process something
}else{
        // process something
}
```

MUTATOR: MATH MUTATOR

- It replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation

Original Conditional	Mutated Conditional
+	-
-	+
*	/
/	*
%	*
<<	>>

```
int a = b + c
```

mutated to

```
int a = b - c
```

```
public class A{
        int i;

        public void foo(){
                this.i++;
        }
}
```

mutated to

```
public class A{
        int i;

        public void foo(){
                this.i = this.i -
1;
        }
}
```

MUTATOR: INCREMENTS MUTATOR

- It mutates increments, decrements and assignment increments and decrements of local variables (stack variables)
- It will replace increments with decrements and vice versa.

```
public int method(int i){
        i++;
        return i;
}
```

mutated to

```
public int method(int i){
        i--;
        return i;
}
```

MUTATOR: VOID METHOD CALL MUTATOR

- It removes call to void methods

```
public void method(int i){
        // do something
}

public int foo(){
        int i = 5;
        method(i);
        return i;
}
```

mutated to

```
public void method(int i){
        // do something
}

public int foo(){
        int i = 5;
        return i;
}
```

Constructor Calls are not void method calls

PITTEST: SETUP AND INSTALLATION

- MAVEN PLUGIN
- GRADLE PLUGIN
- CLI

PITTEST: MAVEN PLUGIN

- Add the following plugin declaration in the plugins section of the pom.xml

```
<plugin>
        <groupId>org.pitest</groupId>
        <artifactId>pitest-maven</artifactId>
        <version>LATEST</version>
</plugin>
```

Use the command to run mutation coverage:
mvn org.pitest:pitest-maven:mutationCoverage

PITTEST : RUNNING THE TEST

For each mutation PITTEST will report one of the following outcomes

- Killed
- Lived
- No coverage
- Non Viable
- Timed Out
- Memory Error
- Run Error

Under normal circumstances one should see no **Non Viable** and **Run Errors**

- Mutation Testing is not 100% bulletproof
- Might return false positives

MUTATION TESTING LESSONS

- Mutation Testing Improves Code Reviews
- Not all Mutations Add Value
- Start Small
- Laziness Creates Surviving Mutants
- Value of Mutation Testing is Not recognized early, so have patience
- 100% Mutation Kill score is possible but expensive
- Mutation Testing can lead you to better programmer

SECURITY TESTING

WHY SECURITY TESTING NECESSARY

Process of assessing and testing a system to discover security risks and vulnerabilities of the system and its data

SECURITY TESTING TYPE

- Vulnerability Assessment
  - System is scanned and analyzed for security issues
- Penetration Testing
  - System undergoes analysis and attack from simulated malicious attackers
- Runtime Testing
  - System undergoes analysis and security testing from an end user
- Code Review
  - System code undergoes a detailed review and analysis looking specifically for security vulnerabilities

PENETRATION TESTING

- Is carried out as if the tester was a malicious external attacker with a goal of
  - breaking into the system and either stealing data or carrying out some sort of denial-of-service attack.
- Fewer false positives (results that report a vulnerability that isn’t actually present),
  - but can be time-consuming to run.
- Used to test defense mechanisms, verify response plans, and confirm security policy adherence.
- Helps to uncover new vulnerabilities as well as regressions for previous vulnerabilities in an environment which quickly changes

PENETRATION TESTING PROCESS

Explore

- Tester attempts to learn about system
- trying to determine software is in use, what endpoints exist, patches installed, etc.
- searching site for hidden content
- searching for known vulnerabilities & other weakness

Attack

- Tester attempts to exploit the known or suspected vulnerabilities to prove they exist

Report

- reporting the vulnerabilities
- way system was exploited
- severity of the exploitation

PENETRATION TESTING : GOALS

- Search for vulnerabilities so that these vulnerabilities can be addressed
- Verifying that the system is not vulnerable to known class or specific defect
- Verifying that vulnerabilities that have been fixed has actually been fixed

PENETRATION TESTING STRATEGIES

- Brute Force
  - Is an exhaustive attack that works by testing every possible value of a parameter (password, file name, etc.)
- Cache Poisoning
  - Is an attack that seeks to introduce false or malicious data into a web cache, normally via HTTP Response Splitting
- DNS Poisoning
  - Is an attack that seeks to introduce false DNS address information into the cache of a DNS server, where it will be served to other users enabling a variety of attacks. (e.g., Phishing)

SECURITY CONCEPTS

Cross Functional Concern

- Do we trust the integrity of request coming in from the user's browser?
- Do we trust the upstream services have done the work to make our data clean and safe?
- Do we trust the connection between the user's browser and our application cannot be tampered?
- Do we trust that the services and data stores we depend on?

Trust is not binary.

Assess

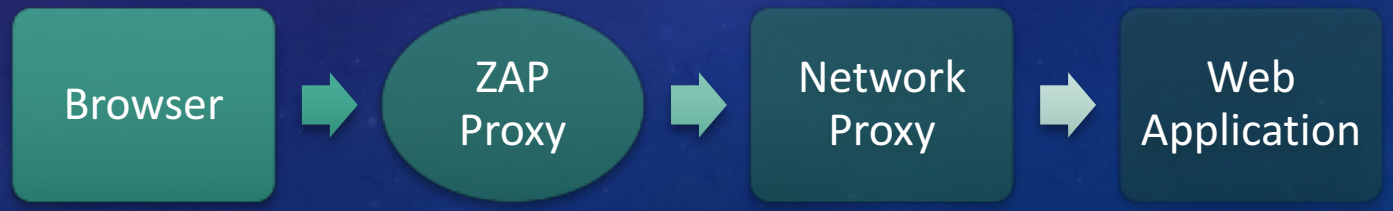
Risk Tolerance → Data Criticality → Investment

SECURITY TESTING TOOL

- FORTIFIER (HP)
- CHECKMARX
- ZAP

PENETRATION TESTING : ZAP

- Zed Attack Proxy (ZAP)
  - Free, open source penetration testing tool maint
  - Designed for testing web application
  - ZAP has at its core “Intercepting Proxy”
  - ZAP intercept and inspect the messages sent between browser and web application
    - modify the contents if needed and then forward those packets on to the destination
    - It mimics the “man in the middle”
  - In corporate environments where another network proxy is in use, ZAP is configured to connect to that proxy



BASELINE SCAN

- Time limited spider of target (default 1 min)
- Just passive scanning
- By default warns on all issues
- Can change to ignore, info or fail
- Can include any ZAP cmdline option
- Can ignore any url regex for any rule

BASELINE SCAN: ISSUES

- All release and beta passive scan rules, eg
  - Missing / incorrect security headers
  - Cookie problems
  - Information / error disclosure
  - Missing CSRF tokens
  - ….

QUICK SCAN

- Quick Scan uses the traditional ZAP spider, which discovers links by examining the HTML in responses from the web application
- This spider is fast, but it is not always effective when exploring an AJAX web application that generates links using JavaScript.
- For AJAX applications, ZAP's AJAX spider is likely to be more effective.
- The AJAX spider is slower than the traditional spider and requires additional configuration for use in a "headless" environment

USING THE ZAP API

- Intro to the API
- Exploring
- Scanning
- Reporting
- Authenticating
- Tuning

INTRODUCTION OF ZAP API

ZAP : EXPLORING

ZAP : PASSIVE SCAN

- Passive scanning happens automatically when proxying
- Passive Scan does not change responses in any way and is considered safe
- Scanning is performed in background thread to not slow down exploration
- Passive Scanning is good at finding some basic vulnerabilities
  - Helps us understand the security state of the application
  - Help us identify or locate area where more investigation are required

ZAP: ACTIVE SCANNING

- Active Scanning finds vulnerabilities by using known attacks against the selected targets
- Active Scan is a real attack on the target and can put the targets at risk
- Never Use Active Scan against targets you do not have permission to test

ZAP : SETUP & INSTALLATION

- ZAP can be installed from the locatio
  execute the installer
- Maven Plugin for ZAP :

```
<plugin>
    <groupId>br.com.softplan.security.zap</groupId>
    <artifactId>zap-maven-plugin</artifactId>
    <version>${zap-maven-plugin.version}</version>
    <configuration>
        <!-- Configuration parameters -->
    </configuration>
    <executions>
        <execution>
            <phase>verify</phase>
            <goals>
                <goal>analyze</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

SECURITY TESTING : STATIC CODE ANALYSIS

One can use tools like Fortifier or CheckMarx to do do static code analysis

CheckMarx and Fortifier Both are commercial product so requires valid license to use it.

PENETRATION TESTING : DEMO

PERFORMANCE TESTING

WHAT IS PERFORMANCE?

- Throughput
- Calculation Time
- Rendering Time
- Page Loading Time
- Memory Usage
- CPU Usage
- Network latency

TYPES OF PERFORMANCE TEST

- LOAD TEST
  - How does the system behave under load? What happens when the load increases from zero to average?
- STRESS TEST
  - What happens when you put an extreme load on the system that the system is not designed to handle?
- SOAK TEST
  - Test the system under moderate/normal load for a longer period of time
- BENCHMARKING
  - Performance regressions / Does the system respond in timely manner?

PERFORMANCE TESTS – WHY SO HARD?

- Simulate production as close as possible:
- Hardware
  - CPU, RAM, storage, …
- Software
  - OS, Virtualization, DBs,...
- Load
- Isolation

Necessary to have:

- Infrastructure
- Monitoring
- Logging

"Without data you're just another person with an opinion" – W. Edwards Deming

WHAT DO WE TEST FOR?

Capacity

- Oversubscribed Resources
  - CPU
  - MEMORY
  - BANDWIDTH
  - STORAGE
  - QUEUES
- Response Time Under Load

Stability

- Resource Leaks
  - Memory
  - Network Connections
  - File Handle
- System Crashes
- System Recovery

PERFORMANCE TESTING : TOOLS

Open Source Tools

- Apache Jmeter
- Gatling (Scala)
- Locust (Python)
- GoReplay (Go)
- Artillery (JavaScript/NodeJs)
- Grinder (Java)
- Siege
- Tsung (Erlang)

Commercial Tools

- Load Runner (HP)
- Load Complete (SmartBear)
- WAPT
- BlazeMeter
- SilkPerformer (MicroFocus)
- ApicaLoadTest

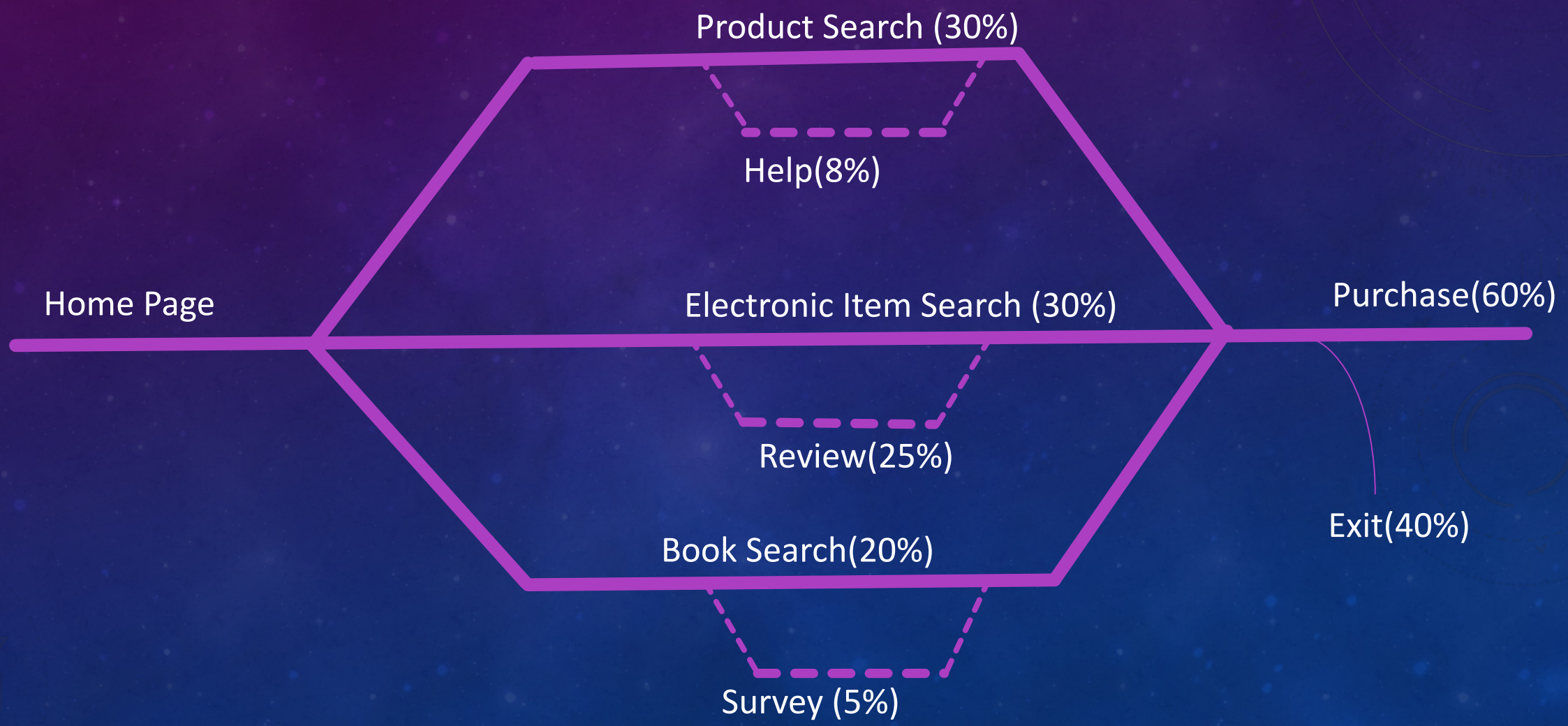
Cloud Solutions

- BlazeMeter
- Flood.io
- Load.io
- LoadStorm
- LoadImpact
- ApicaLoadTest

GET YOUR USER SCENARIOS

- Goal
  - To simulate realistic business behaviors of the users
- What are they doing?
  - Normal/Common/Explicit activities
  - Implicit Activities
  - Critical Activities
  - Resource Hungry activities

SAMPLE SCENARIO



MODEL USER SCENARIO: APPROACH

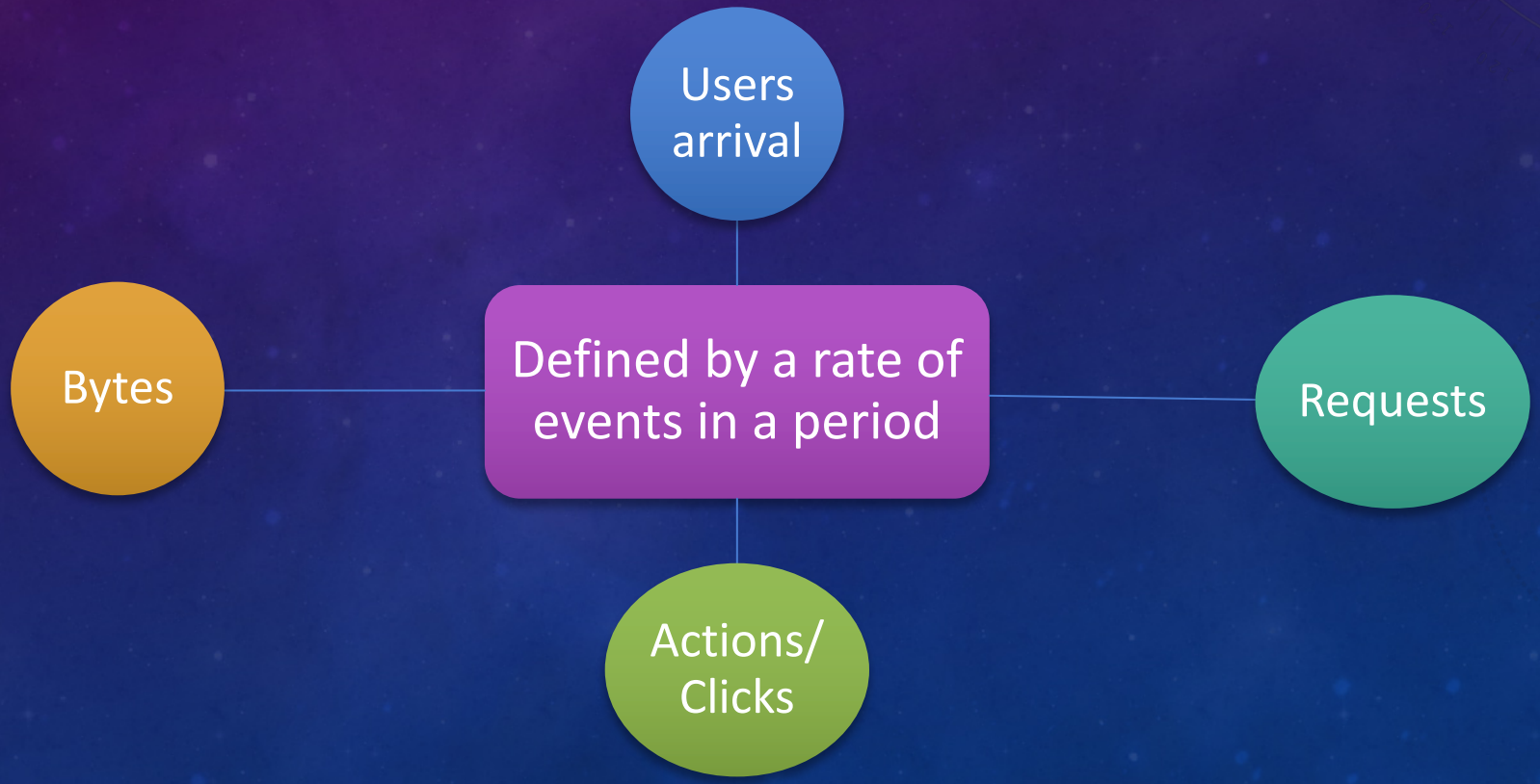
- Criteria
  - Based on the Business goal come up with user scenarios which cover most critical flows
  - What are the response times for different pages?
- Design
  - Identify and Model the scenarios
  - Document the Scenarios
- Analyze
  - Analyze the gatling reports for the response time

LOAD MODELLING

CONCEPTS/STRATEGIES

LOAD MODEL

The Load Model basically defines “How Load is distributed over time?”



EQUI DISTRIBUTION

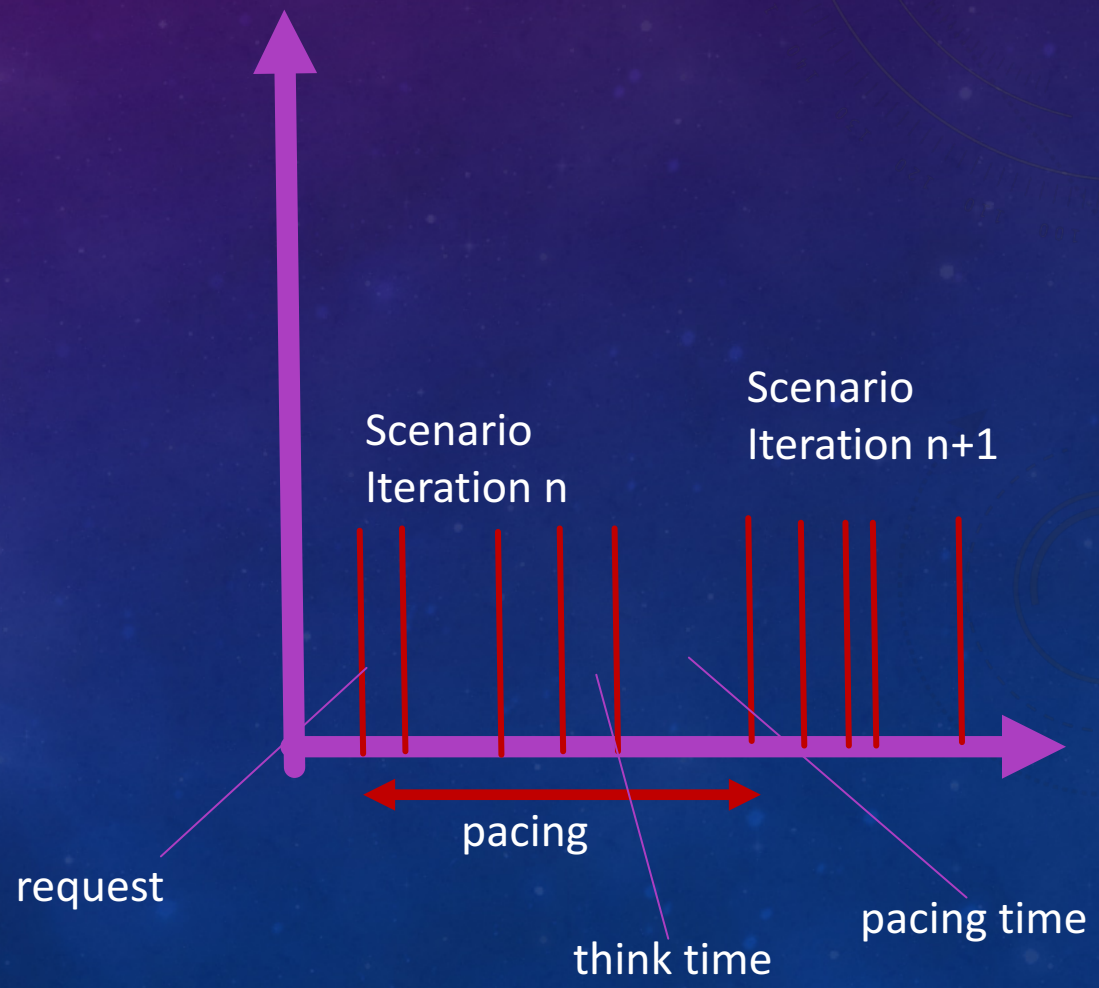
- Completely Synthetic
- Suitable For
  - Long Running Stability Tests
  - Benchmarking
- Load should aim at 80% capacity
- Good for tracking long term effects
  - Resource/Memory Leaks
  - Garbage Collection Singularities
- Native Support exists in Gatling

RAMP UP

- Slowly increasing Load on System
- System can warm-up
  - Populate Caches
  - Compile & Optimize Code (JIT)
- User & Requests get distributed
- Native Support exists in Gatling

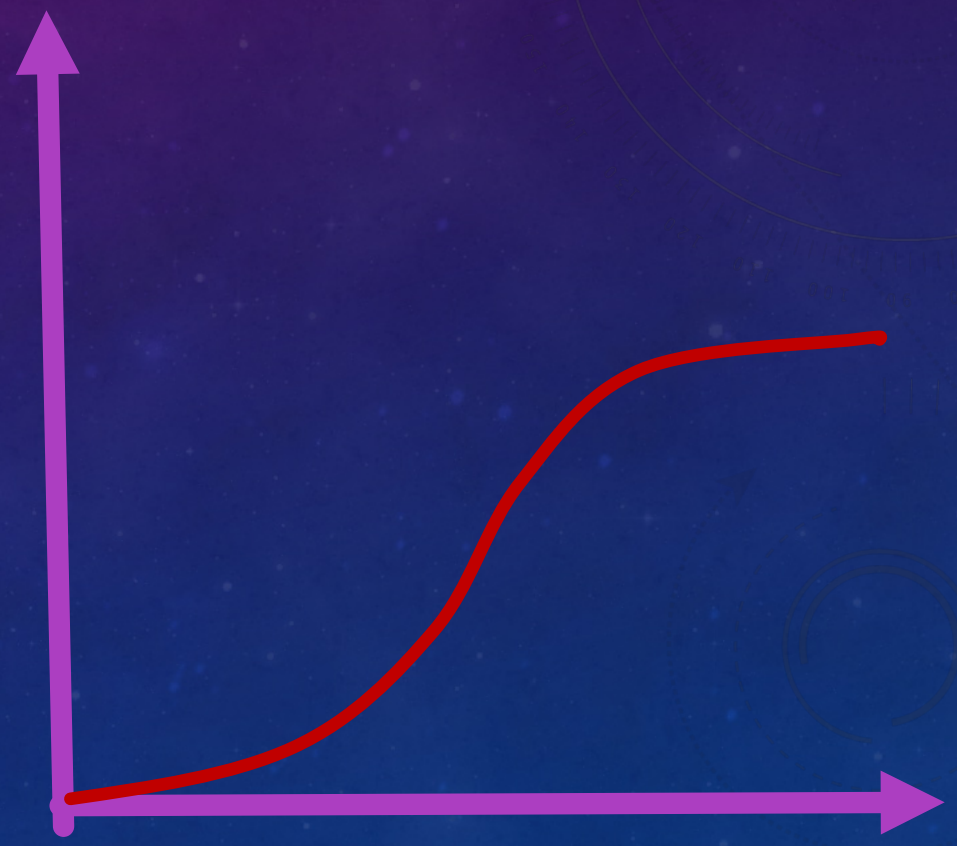
PACING & THINK TIME

- Think Time
  - Time to wait between each request
  - Simulates an actual user behavior
- Pacing
  - Recommended for Equi-Distribution
  - Keep the users and load distributed
  - Minimum time for a user transaction before the next begins
  - Wait time is added at the end



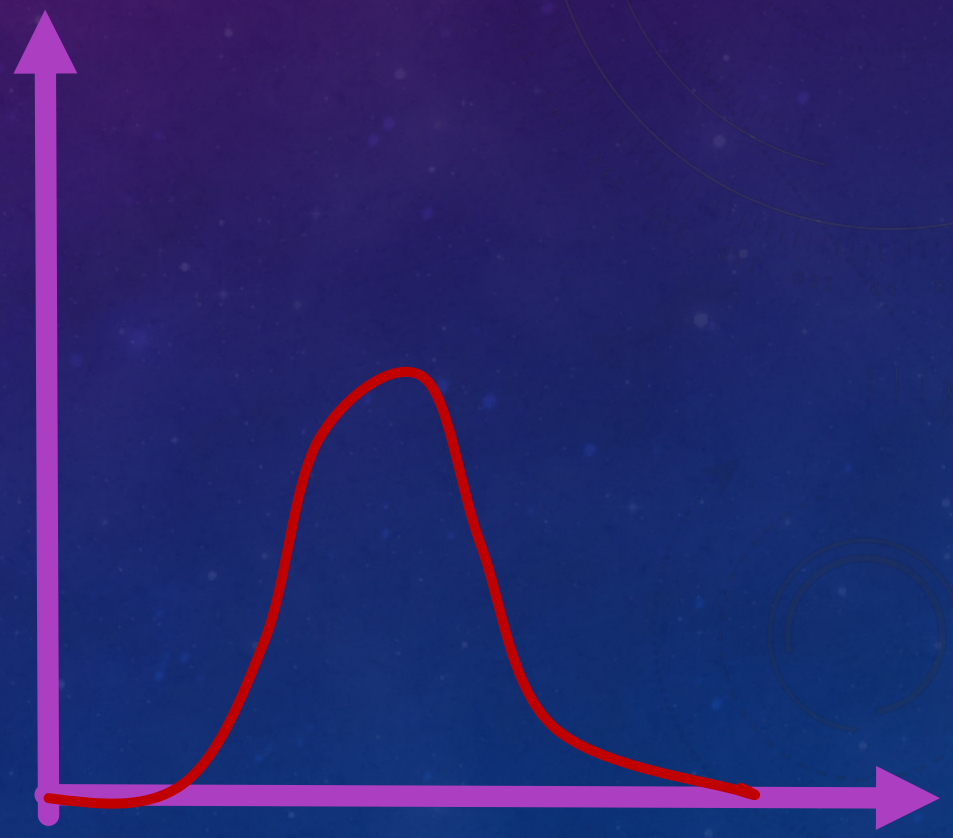
HEAVISIDE DISTRIBUTION

- Smoothed Rampup
- More realistic load increase
- Native support exists in Gatling



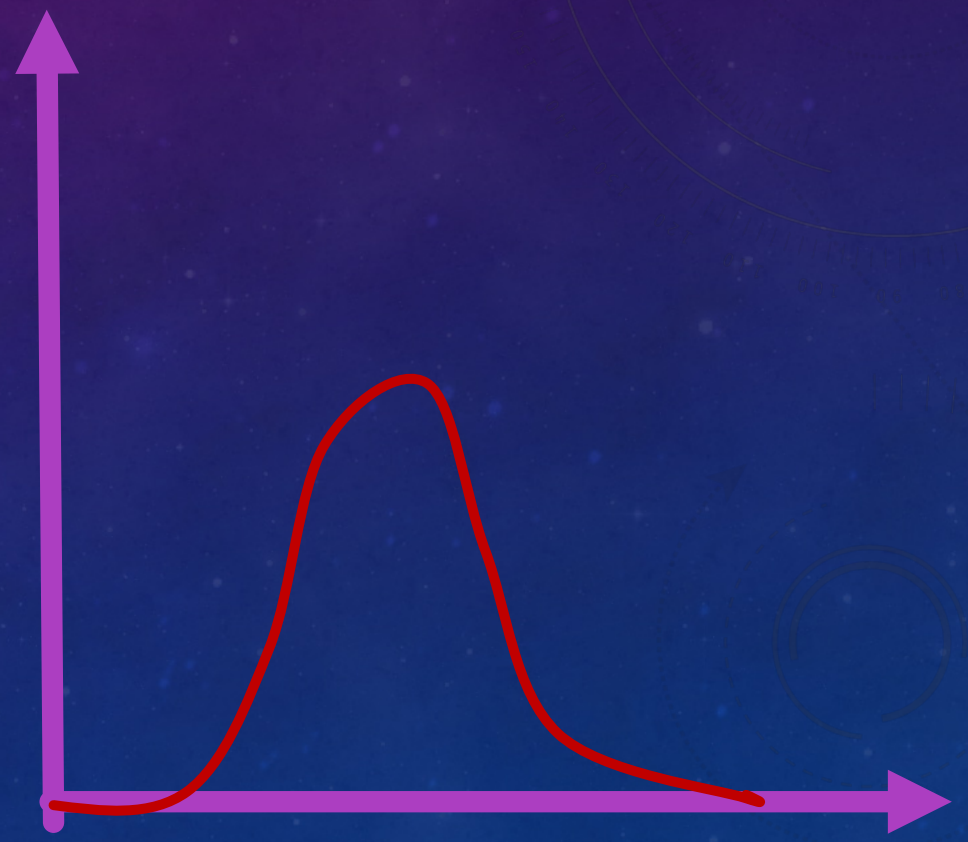
POISSON DISTRIBUTION

- High peak load and long tail
- Resemble arrival times of a singular event like Sales Day, Marketing Campaign
- Good for modelling real load and peak events
- Good for overload events, i.e. Go Live Scenarios
- Native support exists in Gatling



GAUSS/NORMAL DISTRIBUTION

- Peak load with lower leads / tails
- More realistic distribution
- Good for modelling daily loads
- Native support exists in Gatling



SYSTEM THEORY

When to use what?

Closed Loop

- Load generation rate depends on the Response rate
- $R_{load} = func(const, R_{response})$

Open Loop

- Load generation rate is independent of Response rate
- $R_{load} = const$

Closed Loop

- Closed User groups
- Internal Application
- Measure/Validate
  - Capacity
  - Throughput
- Suitable For
  - Load Tests
  - Benchmarking

Open Loop

- Unknown User groups
- Internet Application
- Measure/Validate
  - Throughput
  - Queue Sizes
- Suitable For
  - Load Tests
  - Benchmarking
  - Stress Tests

PERFORMANCE TESTING : JMETER

JMETER : DEMO

SONARQUBE

Static Code Analysis - Sonarqube

- Enables Continuous Code Quality Inspection
- Code analyzer for 20+ programming language (Java, PI/SQL, Python, CSS…)
- Used by >85K organization across globe
- Reports code quality validations:
  - Bugs
  - Code Smells
  - Vulnerabilities
- Provide details about code coverage, duplicate lines, code complexities etc.

QAI

Sonarqube Cont…

- **Bug:** An issue that represents something wrong in the code. If this has not broken yet, it will, and probably at the worst possible moment. it includes demonstrably wrong code, or code that is more likely not giving the intended behaviour.

- **Code Smell:** A maintainability-related issue in the code. “Smelly” code does (probably) what it should, but it will be difficult to maintain. In the worst cases, it will be so confusing that maintainers can inadvertently introduce bugs. Examples include **duplicated code, uncovered code by unit tests and too complex code**.

- **Vulnerability:** A security-related issue which represents a backdoor for attackers. Examples include SQL injection, hard-coded passwords and badly managed errors

QAI

Sonarqube Cont…

http://vineshs-macbook-air.local:9000/

- **Quality Rules**
- **Quality Profiles**
- **Quality Gates**

QAI

Sonarqube Cont…

Sonar-Scanner

- **Download: https://binaries.sonarsource.com/Distribution/**
- **Set up server location in configuration (***<install_directory>/conf/sonar-scanner.properties)*
- **Add the** *<install_directory>/bin* **directory to your path**
- **Create Project's properties file(**sonar-*project.properties***)**
- **Execute command "sonar-scanner" from project's root**

```
sonar.projectKey=com.tekmentors.app:TDDApp
# this is the name and version displayed in the SonarQube UI. Was mandat
sonar.projectName=tdd-app
sonar.projectVersion=1.0

# Path is relative to the sonar-project.properties file. Replace "\" by
# This property is optional if sonar.modules is set.
sonar.sources=src/main/java/
sonar.junit.reportPaths=target/surefire-reports/
 sonar.java.binaries=target/classes/
# Encoding of the source code. Default is default system encoding
#sonar.sourceEncoding=UTF-8
```

```
#----- Default SonarQube server
sonar.host.url=http://localhost:9000

#----- Default source code encoding
#sonar.sourceEncoding=UTF-8
```

QAI

Sonarqube Cont…

sonarqube | Projects | Issues | Rules | Quality Profiles | Quality Gates | Administration | Search for projects, sub-projects and files... | A

My Favorites | All

Perspective: Overall Status | Sort by: Name | Search by project name or key | 1 projects

Filters

Quality Gate

Passed	1
Warning	0
Failed	0

Reliability (Bugs)

A	1
B	0
C	0
D	0
E	0

Security (Vulnerabilities)

A	1
B	0
C	0
D	0
E	0

Maintainability (Code Smells)

A	1

tdd-app Passed

Last analysis: November 12, 2018, 8:26 PM

0 A	0 A	9 A	75.0%	0.0%	262 XS
Bugs	Vulnerabilities	Code Smells	Coverage	Duplications	Java

1 of 1 shown

QAI

SONARQUBE INSTALLATION

- https://www.sonarqube.org/downloads/
- https://www.sonarsource.com/products/codeanalyzers/sonarpython.html
- https://www.sonarsource.com/products/codeanalyzers/sonarcsharp.html
- sudo apt install openjdk-8-jdk

- Go to the **Administration** tab -> **Marketplace** -> **Installed**
- Confirm that SonarPython plug-in is installed, if not install it.
- Restart the SonarQube server if needed.

DAY 2

Day 02 – The Big Picture

DevOps Tools Introduction and Live demo

- **SCM : Live Demo** (Hands on Lab by students)
  - Understanding Basics of Git
  - Github – Create an account and fork your application code
  - Git clone the github code , Use maven to compile and package java source code
  - Branching Strategy
- CI/CD Principles

DevOps Tools Practical

- **Defining and Implementing Commit Stage Pipeline**
  - Jenkins Configuration
  - Jenkins Pipeline Concept
  - Configuring Testing Pipeline/Reports View (On Jenkins)
  - Configuring Static Code Analysis Configuration (On Jenkins)
  - Configuring Code Coverage Analysis
  - Integration With SonarQube
- **Configuring SonarQube Build**
  - Configuring SonarQube rules For Java

QAI

GIT

GIT ARCHITECTURE

- Distributed Source Repository
- It is optimized for
  - Distributed Development
  - Large file Counts
  - Complex Merges
  - Enables ease in creation & management of branches
  - Being very fast
  - Being robust

Not optimized for

- Tracking file permissions and ownership
- Tracking individual files with separate history

COMPARATIVE ANALYSIS

Centralized Version System

- Central server keeps the only copy of repository
- Merging/Branching is more challenging
- Not typically suited for the distributed mode of development
- Need to have constant connectivity with the Central server to commit or version your code
- Versioning is Incremental

Distributed Version System

- Entire replica of repository lies on individual developer system
- Most of the operations which developer takes are only applied at the local repository
- Push/Pull/Clone/Fetch are the only operations applied at the remote repository
- Merging/Branching is much easier
- Versioning is not incremental

GIT CONCEPTS

REPOSITORY

Local Repository

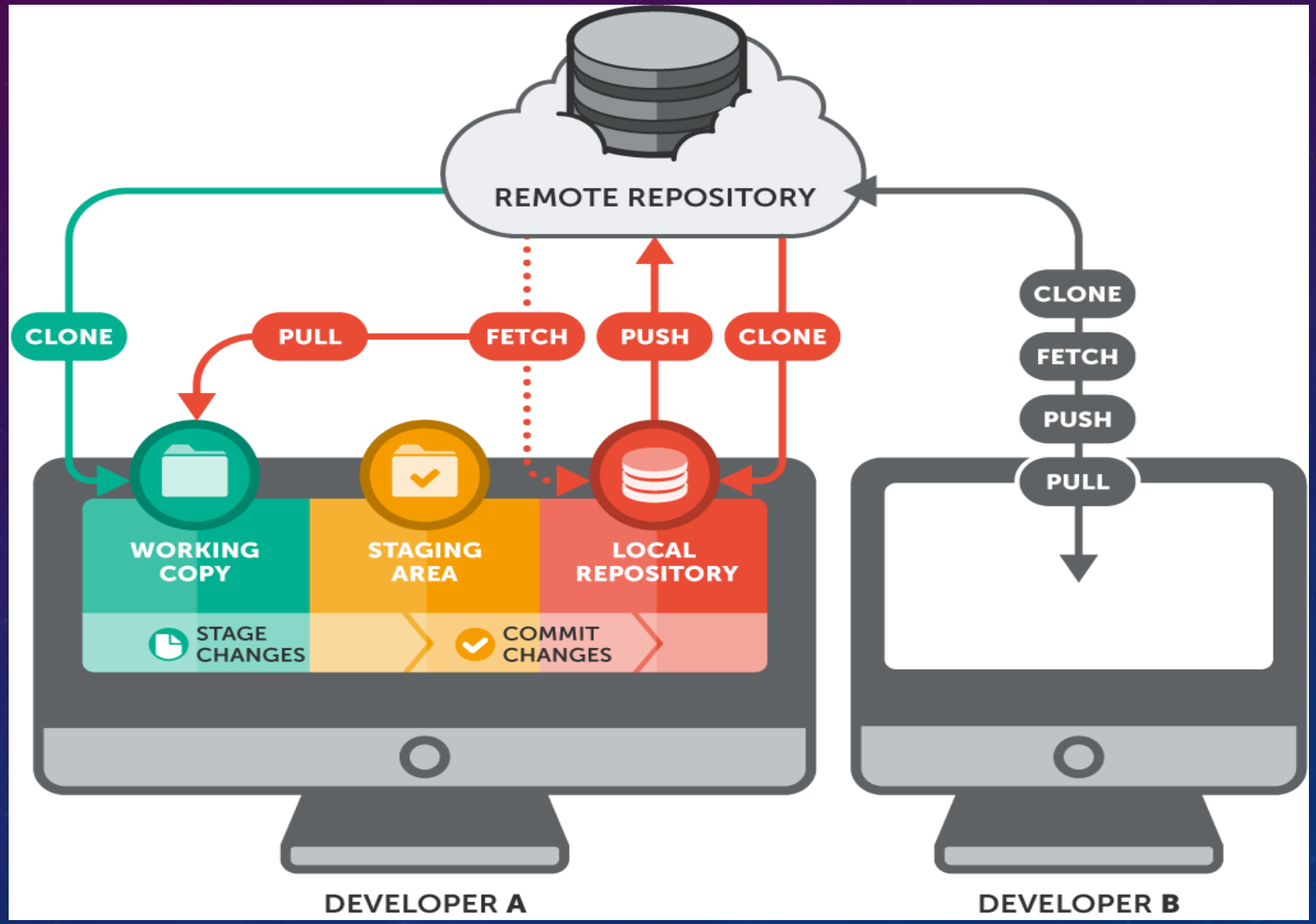
- Resides on local system as a **.git** folder inside project's root folder
- Working Copy resides only in Local Repository
- Actual work on project happens here

Remote Repository

- Located on the remote server
- No Working Copy concept
- It has no working directory but it exclusively consists of the **.git** repository folder
- Not meant for actual working like creation/modification of files
- Is used to share and exchange data between team members

We can have only one working copy with one specific version (not multiple in parallel)

REMOTE VS LOCAL REPOSITORIES

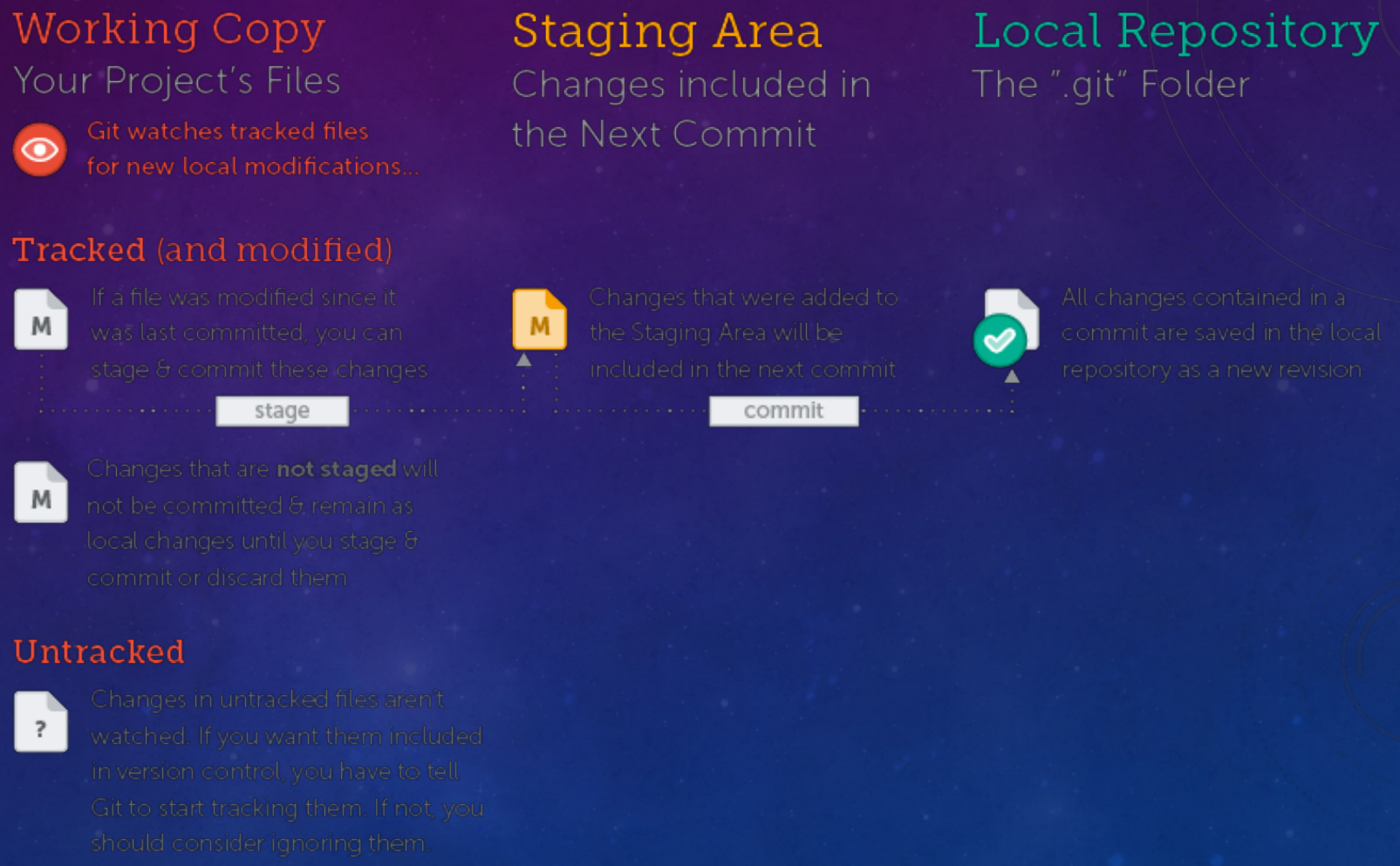


STAGING AREA

- Developers make local changes in the working directory of git repository
- Now developer may decide which changes he wants to include in the next set of commit
- Developer usually does this by adding them to the so-called “Staging Area”

WORKING ON PROJECT

- File Status
  - Untracked
  - Tracked



WORKING WITH GIT

CREATING A REPO

- Creates a .git in the current dir
- Optional: edit ***.gitignore***

```
$git init
```

- Adds all files except ***.git***

```
$git add .
```

- Commits the initial code
- Creates current branch named "**master**"

```
$git commit –m "Message"
```

- Tag the initial code base

```
$git tag "Initial Codebase"
```

- Push the code to remote server

```
$git push origin master --tags
```

IMPORTING A NEW PROJECT

- Assume you have tarball with your initial work

```
$ tar xzf project.tar.gz
$ cd project
$ git init
```

- Git will reply

```
Initialized empty Git repository in .git/
```

- Take a snapshot of contents of all files under current directory

```
$git add .
```

- Snapshot is now stored in a temporary area which Git calls the “index”
- Permanently store the contents of the index in the repository with

```
$git commit –m “Message”
```

CLONING

- Creates a git repo from an existing repo
- Generally creates a subdirectory
- Your workfiles and .git are in there
- Remote branches are “tracked”
- Remote “HEAD” branch checked out as your initial “master” branch as well
- Clone repo identified as “origin”

```
$git clone <repo_url>
```

GITIGNORE

Files which we do not want to version control has to be explicitly mentioned in the .gitignore files

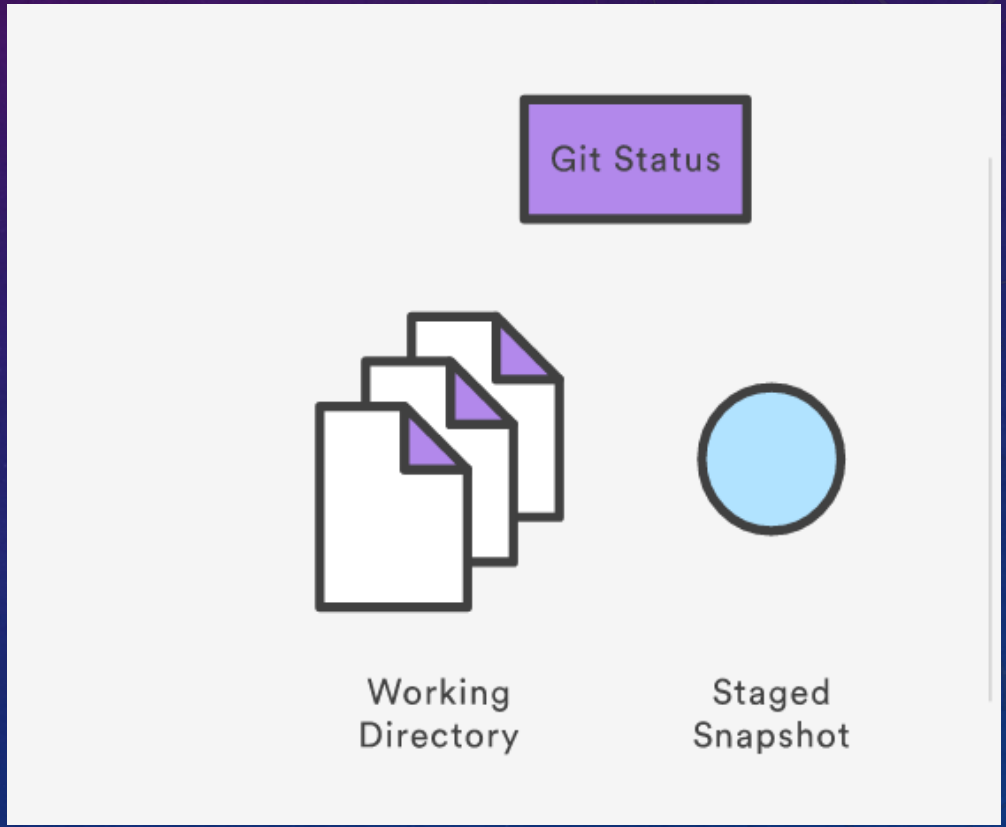
As a rule of thumb we want to ignore files that were automatically created like temporary files, cache files, logs, compiled files…

- **Ignore one specific file**: Provide the full path to the file, seen from the root folder of your project.
  path/to/file.ext
- **Ignore all files with a certain name** (anywhere in the project): Just write down the file's name, without giving a path.
  filename.ext
- **Ignore all files of a certain type** (anywhere in the project):
  *.ext
- **Ignore all files in a certain folder**:
  path/to/folder/*

INSPECTING A REPOSITORY

<u>git status</u>

- Displays the state of the working directory & staging area
- Enable to see staged/unstaged changes
- Enable to see state of tracked/untracked files
- Does *not* show you any information regarding the committed project history

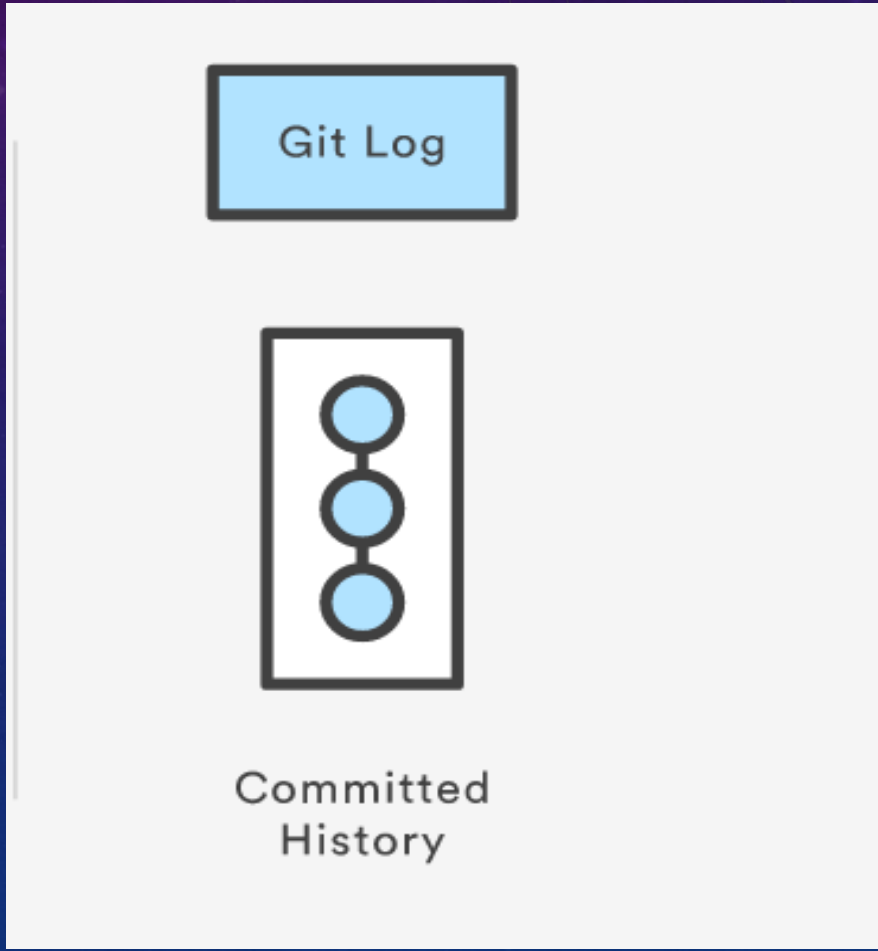


INSPECTING A REPOSITORY

<u>git log</u>

- Displays committed snapshots

- Enable to see project history, filter it, and search for specific changes



git status let you inspect the working directory and staging area while git log only operates on committed history

GIT LOG

Usage

$git log	Entire Commit history
$git log –n 3	Display only 3 commits
$git log --oneline	Condenses each commit to a single line
$git log --stat	Along with git log information, include which files were altered and relative number of files added or deleted from each of them
$git log -p	shows the full diff of each commit
$git log --author="<pattern>"	Search for commits by a particular author
$git log --grep="<pattern>"	Search for commits with commit message by a specific pattern

GIT LOG

Usage

```
$git log <since>..<until>
```

Shows commits that occurred between two parameters
Both the arguments can be either a ***commit ID***, ***a branch name, HEAD***

```
$git log <file>
```

Only display commits that include the specified file

```
$git log --graph --decorate --online
```

--graph flag will draw a text based graph of the commits on the left hand side of the commit message
--decorate adds the names of branches or tags of the commits that are shown

UNDOING CHANGES

- git checkout
- git revert
- git reset
- git clean

GIT CHECKOUT

Checking out files

Checking out commits

Checking out branches

Checking Out Commits

- Makes the entire working directory match that commit
- This can be used to view an old state of your project without altering the current state in any way

```
$git checkout commitId
```

Checking Out File

- This can be used to view to see an old version of that particular file, leaving the rest of the working directory untouched

```
$git checkout commitId file
```

GIT REVERT

git revert command undoes a committed snapshot.

```
$git revert <commitId>
```

RESET

git reset is a simple way to undo changes that haven't been shared with anyone else

```
$git checkout hotfix
$git reset HEAD~2
```

EXPLANATION OF GIT RESET SOFT AND HARD

- Say if we are on branch **master** with the series of commits

HEAD points to C
INDEX points to C

```
$git reset --soft B
```

HEAD now points to B, but the INDEX still points to C

```
$git status
```

will show them as staged
so if we do commit, we will get a new commit with same changes as C

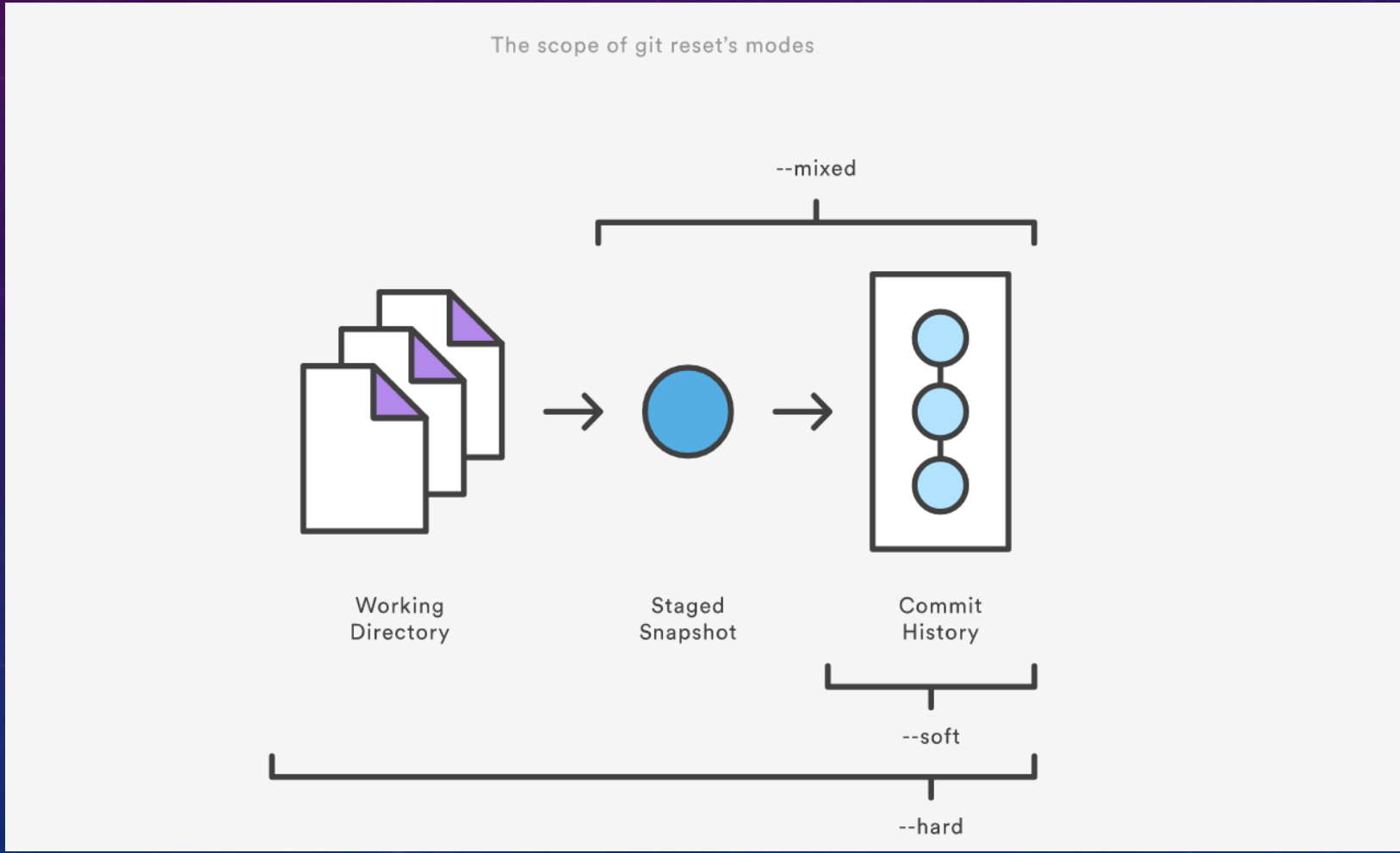
```
$git reset --hard B
```

HEAD now points to B, and the INDEX also points to B

```
$git status
```

will show them as unstaged

RESET SCOPE


The scope of git reset's modes
--mixed
Working Directory
Staged Snapshot
Commit History
--soft
--hard

GIT RESET USAGE

$git reset file	Remove the specific file from staging area, but leave working directory unchanged
$git reset	Reset the staging area to match the most recent commit
$git reset --hard	Reset the staging area to match the most recent commit as well as working directory
$git reset commitId	Move the current branch tip backward to commitId
$git reset --hard commitId	Move the current branch tip backward to commitId as well as the working directory

UNDO SCENARIOS

Undo Add

```
$ edit (1)
$ git add file1.txt file2.txt
```

Somebody asked you to pull, and the changes sound worthy of merging

- By this time I have already corrupted my index (i.e. my index does not match the HEAD) commit
- As the remote changes are not going to affect file1 and file2.txt so you revert the index changes for these two files

```
$ git reset
$ git pull
```

file1.txt and file2.txt still remains in the working copy

UNDO SCENARIOS

Undo Commit And Redo

```
$ git commit –m “Message”
```

You realized that commit was erroneous or incomplete

```
$ git reset --soft HEAD^
```

Revert the working tree to state it was before commit
reset copies the old head to .git/ORIG_HEAD

- Go edit the files and once satisfied with the change recommit again

```
$ git commit –m “Message”
```

If you do not need to edit the message further you can choose the –C option

```
$ git commit  -a –c ORIG_HEAD
```

UNDO SCENARIOS

Undo Commit And Making it a Feature/Topic Branch

You have made some commits on **master** branch and realized that they were premature to be in the **master** branch

You want your code to be part of some feature branch, so create feature branch from the current HEAD

```
$ git branch feature/account_mgmt
```

Created a feature branch

```
$ git reset --hard HEAD~3
```

Rewind the master branch to get rid of those ***three commits***

- Checkout the feature branch and starts working on it

```
$ git checkout feature/account_mgmt
```

UNDO SCENARIOS

Undo Delete

You have made some commits on **master** branch and after that by mistake you deleted some files permanently

You want your code file which you deleted to be reverted back to the working copy

```
$ delete files
```

Delete the files

```
$ git status
```

It will give you some error message

- To revert back the original files, get the last commit id using following command

```
$ git log
```

```
$ git reset --hard commit_id
```

UNDO SCENARIOS

Undo Merge or Pull

Trying to update from the upstream resulted in a lot of conflicts. You were not ready to spend a lot of time merging right now, so you decide to do that later

```
$ git pull
```

Resulted in lot of conflicts

```
$ git reset --hard
```

Clears the mess from index file and the working tree

- Merge a topic/feature branch into the current branch

```
$ git pull .   topic/branch
```

EXPLORING HISTORY

- Git history is represented as a series of interrelated commits

```
$git log
commit c82a22c39cbc32576f64f5c6b3f24b99ea8149c7
Author: Arun Tiwari<arunstiwari@gmail.com>
Date:   Mon July17 17:18:22 2006 -0700
```

- To see the detail of the commit

```
$git show c82a22c39cbc32576f64f5c6b3f24b99ea8149c7

$git show HEAD //tip of the current branch

$git show develop // tip of the develop branch
```

- Every commit usually has one parent

```
$git show HEAD^        #To see the parent of the HEAD

$git show HEAD^^       #To see the grandparent of the HEAD

$git show HEAD~4       #To see the great grandparent of the HEAD
```

PULL

- Pull command performs two operations
  - It fetches changes from a remote branch
  - It merges the changes into current branch

SCENARIO AROUND PULL/FETCH

- X has started a new project with Git repository in /home/X/project
- Y clones the repository git clone /home/X/project myrepo
- Y makes some changes and commits them

  (edit files)

  Y$ git commit -a

  (repeat as necessary)

- Y is ready and he asks X to pull the changes from his repository /home/Y/myrepo
- X does this with git pull /home/Y/myrepo master

- X can peek at what Y did without merging first, using “fetch” command
- This allow X to inspect what Y did using a special symbol ”FETCH_HEAD” in order to determine if he has anything worth pulling, like this

  Y$git fetch /home/Y/myrepo master

  Y$git log –p HEAD..FETCH_HEAD

HOW TO RESOLVE MERGE CONFLICTS

- On seeing the conflict, you can do two things
  - Decide Not to Merge : Just use **git merge --abort**
  - Resolve the Conflicts
    - Manually resolve the conflicts and save the file
    - Add them to the index using **git add filename**
    - Commit the files **git commit –m “message”**

DEMO

Repository Creation Demo

GIT BRANCH

BRANCH

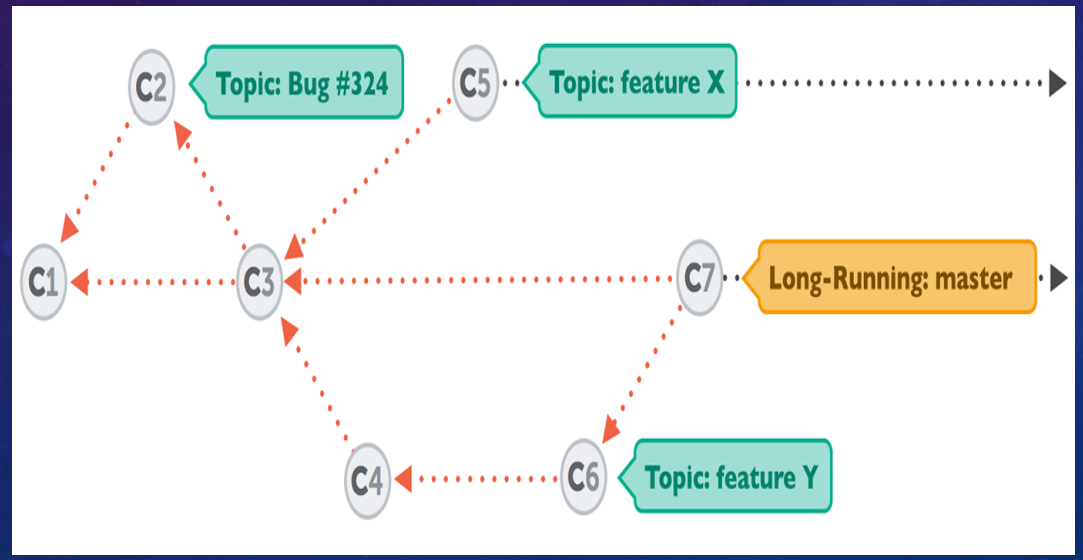
- Default branch in case of git is master branch
- Branches always have a context

Example

- **master** : Here context is it reflects the production state
- **develop:** Here the context is it reflects the integration code base
- **feature-name:** Context of the given feature development
- **hotfix-xxx :** Hot fix branch for version xxx
- **release-xxx:** Release branch for version xxx

BRANCHING WORKFLOWS

- Short Lived/Topic/Feature Branch
  - They are about a single topic.
  - Used to isolate code belonging to this topic from any other code
  - Never create a say “Shopping Cart” branch and commit code dealing with “newsletter signup”
  - It is short-lived and meant to live till the feature or topic is completed
  - Merge the topic branch code to broader context branch like development before deleting it



- Long Running Branch
  - Like master/development
  - Should not work on them directly
  - Integrate Other branches code into them
  - Long running branches can have hierarchy between them e.g.
    - master is used as the highest-order branch
    - development can be subordinate branch

CREATING A BRANCH

1. Checkout the branch from which you want to create a new branch

```
$git checkout develop
```

2. Create a branch from the develop branch using following command

```
$git checkout –b feature-1.1.0 develop
```

feature-1.1.0 *is new feature name*

MERGING A BRANCH

1. Checkout the branch on which you want to merge

```
$git checkout develop
```

2. Merge the feature branch into develop branch using following command

```
$git merge –no-ff feature-1.1.0
```

feature-1.1.0 *is branch to be merged*

3. You can then push the merged features of develop into remote repository

```
$git push origin develop
```

4. You can delete the feature branch at some stage using the following command

```
$git branch -d feature-1.1.0
```

MERGE IS BETTER IN GIT

- Merge is local
- git knows the ancestry
- powerful merge strategies

MERGE STRATEGIES

GIT has breadth of choices on how to merge changes

- resolve
- recursive
- octopus
- ours
- subtree
- yours?

What is a fast-forward merge?
It will just shift the HEAD tag

WHAT IS REBASE?

It is a way to replay commits, one by one, on top of a branch

- Interactive Rebase: Helps you clean up your private branches before publishing them
- Rebase rewrites history
  - Treat this power with great care. Only rewrite history of local branches or …

COMPARISON BETWEEN MERGE & REBASE

Merge Commit	Rebase (FF)	Rebase (SQUASH)
Ugly History	No merge commits	Easy to read
Full traceability	Verbose History	Can be more difficult to trace changes
Hard to screw up		

GIT STASH

- A commit wraps up changes and saves them permanently in the repository
- There are situations when you want to save the local changes temporarily
  - Consider you are in middle of some changes for feature X when an important bug report comes in
  - Your local changes does not belong to the bugfix you are going to fix
  - You want to save those changes temporarily and checkout the bug fix branch to work on bug

Git Stash feature helps you save your changes temporarily and then having a clean working copy to make bug fix

GIT STASH

Stashing your work

```
$ git status
# On branch master
# Changes to be committed:
#           (use "git reset HEAD <file>..." to unstage)
#
#           modified: index.html
#
# Changes not staged for commit:
#           (use "git add <file>..." to update what will be
committed)
#
#           modified: lib/simplegit.rb
#
```

```
$ git stash
Saved working directory and index state \ "WIP on
master: 049d078 added the index file" HEAD is now at
049d078 added the index file (To restore them type
"git stash apply")
```

Now Working Copy Clean

```
$ git status
# On branch master nothing to commit, working
directory clean
```

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

```
git stash apply stash@{2}.
```

```
git stash drop stash@{2}.
```

WHEN TO STASH

- Stash enables us to get a clean working copy
- Apply Stash under following circuminstances
  - Before checking out a different branch
  - Before pulling remote changes
  - Before merging or rebasing a branch

SOME ADDITIONAL COMMANDS

```
$git log –graph –color –decorate –oneline --all
```

```
* 6ae58c6 (origin/F-1.2.0, F-1.2.0) Added rollback and testdata along with some feature specific files
* d33ae6a Added the Insert Record scripts for Feature 1.2.0
| * 7e06634 (HEAD -> F-1.2.1, origin/F-1.2.1) Added Implementation For Feature 1.2.1
|/
* 1e8de27 (origin/master, origin/develop, master, develop) Added additional files
* c68acd9 (tag: Initial) Initial Database Code
```

```
$git show-branch --all
```

```
! [F-1.2.0] Added rollback and testdata along with some feature specific files
 * [F-1.2.1] Added Implementation For Feature 1.2.1
  ! [develop] Added additional files
   ! [master] Added additional files
    ! [origin/F-1.2.0] Added rollback and testdata along with some feature specific files
     ! [origin/F-1.2.1] Added Implementation For Feature 1.2.1
      ! [origin/develop] Added additional files
       ! [origin/master] Added additional files
--------
+   +    [F-1.2.0] Added rollback and testdata along with some feature specific files
+   +    [F-1.2.0^] Added the Insert Record scripts for Feature 1.2.0
 *   +   [F-1.2.1] Added Implementation For Feature 1.2.1
+*++++++ [develop] Added additional files
```

```
$git log master..F.1.2.1 --oneline | tail -1
7e06634 Added Implementation For Feature 1.2.1
```

SOME ADDITIONAL COMMANDS

```
$git diff --cached
```

```
$git diff
$git status
```

For unstaging files use following command

```
$git reset HEAD file2.txt
```

```
$git tag <tagName> <commit_id>
$git diff tagName HEAD
$git branch branchName tagName
```

COMMITS RULE THE REPO

- One or more commits form the head of object chains
- Typically one head called “master”
- Others can be made at will (“branches”)
- Usually one commit in the repo that has no parent commit (“root” commit)

HOW DOES GIT DO IT?

- Universal public identifiers
  - None of the SVK “my@245 is your @992”
- Multi protocol transport : HTTP, SSH, GIT
- Efficient object storage
  - Everyone has entire repo (disk is cheap)
- Easy branching and merging
  - Common ancestors are computable
- Patches (and repo updates) can be transported or mailed
- Binary “patches” are supported

SHA1

- Every “object” has a SHA1 to uniquely identify it
  - “objects” consist of:
    - Blobs (the contents of a file)
    - Trees (directories of blobs or other trees)
    - Commits:
      - A tree
      - Plus zero or more parent commits
      - Plus a message about why
    - And tags

TAGS

- An object (usually a commit)
- Plus an optional subject (if anything else is given)
- Plus an optional payload to sign it off
- Plus an optional gpg signature
- Designed to be immobile
  - Changes not tracked during cloning
  - Use a branch if you want to move around

OBJECTS LIVE IN THE REPO

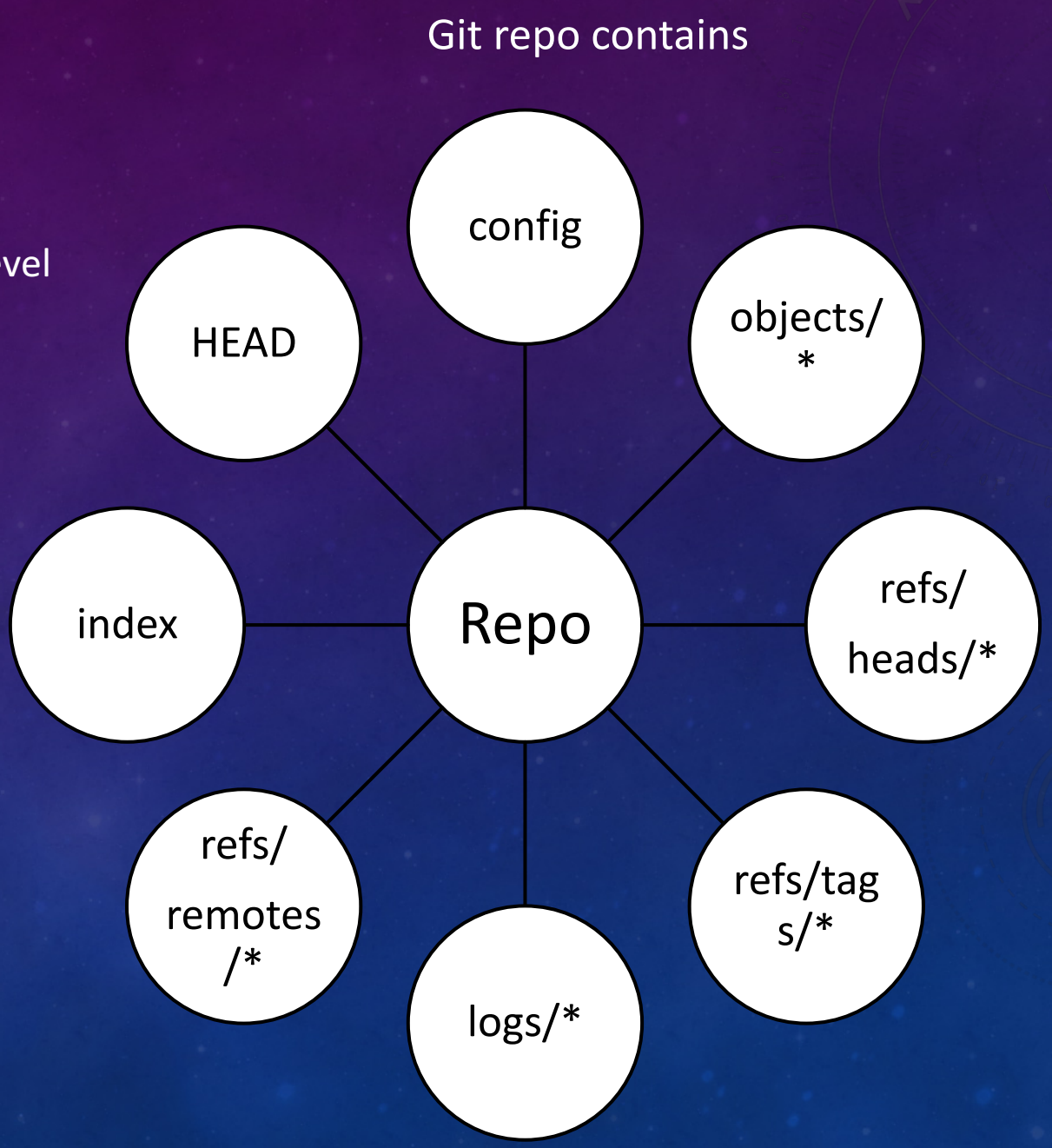
- Git efficiently creates new objects
- Objects are generally added, not destroyed
- Unreferenced objects will garbage collect
- Objects start “loose”, but can be “packed”
- “Packs” represent objects as deltas
- “Packs” are also created for repo transfer

REACHING OUT

- From a commit, reaching the components:
  - Chase down the tree object to get to directories and files as they existed at this commit time
  - Chase down the parent objects to get to earlier commits and their respective trees
- Do this recursively, and you have all of history
  - And the SHA1 depends on all of that!

THE GIT REPO

- A “working tree” has a “.git” dir at the top level
- Unlike CVS, SVN: no pollution of deeper directories
  - This makes it friendly to recursive greps



THE INDEX (OR “CACHE”)

- A directory of blob objects
- Represents the “next commit”
- “Add files” to put current contents in
- “Commit” takes the current index and makes it a real commit object

- Diff between HEAD and index:
  - changed things not yet committed
- Diff between index and working dir:
  - changed things not yet added
  - untracked things

RULES OF VERSION CONTROL

- Commit only Related changes
  - Never mix up changes from multiple, different topics in a single commit
  - Mixed file commit lead to make it hard to understand the change related to the topic
  - Roll backing a specific change also becomes difficult
- Write good commit messages
- Use branches extensively
- Never commit Half-Done Work

WHAT MAKES A GOOD COMMIT

- Related Changes
- Completed work
- Tested Work
- Short and Descriptive Messages

COMMIT CONTENTS

Every commit item consists of the following metadata:

- Commit Hash
- Author Name
- Date
- Commit Message

Commit Hash

- Every commit has a unique identifier: 40 character checksum called the “commit hash”

Generally in Centralized Version Control System ascending revision number is used for this but that is not possible for any distributed Version Control System

PORCELAIN AND PLUMBING

- Low-level git operations are called **“plumbing”**
- Higher level actions are called **“porcelain”**
- The git distribution includes both

Porcelain

- Use from command line
- Don’t Script It
- Future releases might change things

Plumbing

- Intended to be upward compatible

COMMITTING

- Your work product is more commits
- These are always on a “branch”
- A branch is just a named commit
- When you commit, the former branch head becomes the parent
- The branch head moves to be the new commit
- Thus, you’re creating a directed acyclic graph
  - ... rooted in branch heads
- A merge is just a commit with multiple parents

TYPICAL WORKFLOW

- Edit edit edit
- git add files/you/have changed/now
  - This adds the files to the index
  - “git add .” for adding all interesting files
- git status
  - Tells you the difference between HEAD, index, and working directory

FREQUENTLY USED COMMANDS

- git pull origin development
- git checkout –b branchName
- git commit –am “Commit message”
- git merge –no –ff feature-branch
- git rebase
- git branch –d
- git push origin development

WHY DO I NEED A BRANCHING STRATEGY?

- Frequent overwriting of commits of one developer by other
- Flexibility to roll out only given branch
- Difficult to debug what went wrong from the clumsy tree structure
- Situation can get out of control
- We need a stable branch which is ready to be deployed to production

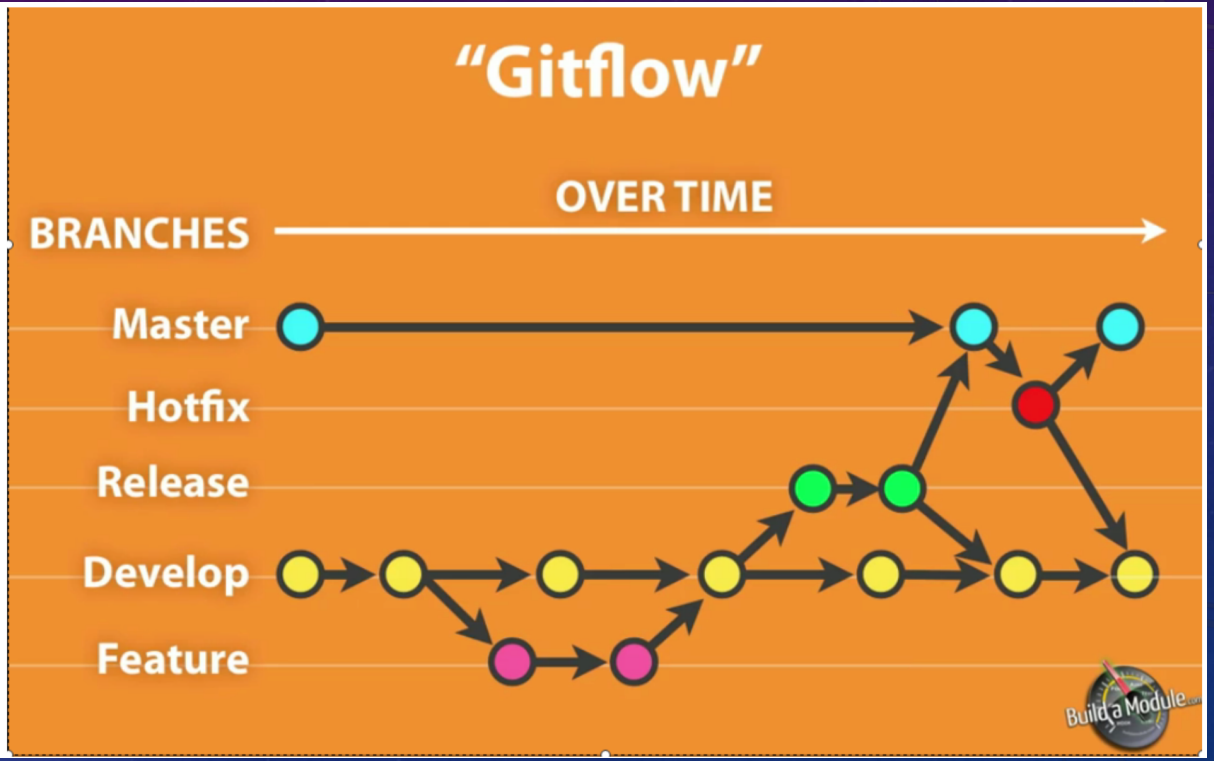
BRANCHING STRATEGY : GIT-FLOW

- Git flow is a way to manage our code such that
  - There are no code override by my colleague
  - If something goes wrong I can easily figure out which commit failed by referring to a very simple tree

WHAT IS GIT FLOW?

Benefits:

- Parallel Development
- Collaboration
- Release Staging area
- Support For Emerging Fixes



TYPES OF BRANCHES?

- Development branch
- Feature branch (new features, non-emergency bug fixes)
- Release branch
- Master branch
- Hotfix branch

GIT : MAIN BRANCHES

- Central Repository holds two main branches with an infinite lifetime
  - **master**
  - **develop**
- **origin/master** : always reflects a production-ready state
- **origin/develop** : always reflects a state with the latest delivered development changes for the next release (Commonly also referred as ***Integration Branch***)

GIT : SUPPORTING BRANCHES

- Supporting branches
  - enable parallel development between team members
  - ease tracking of features
  - prepare for production releases
  - and to assist in quickly fixing the live production problems
- These branches have a limited life time and is always removed at an appropriate time
- Supporting Branches includes
  - Feature Branches
  - Release Branches
  - Hotfix Branches

Create a Feature Branch off From the develop branch

git checkout –b myfeature develop

Incorporating a finished feature on develop

1. git checkout develop
2. git merge –no-ff myfeature
3. git branch –d myfeature
4. git push origin develop

RELEASE BRANCH

Usually branches off from **develop**

Must merge back into **develop** and **master**

Branch naming convention **release-***

- Support preparation of new production release
- Allow for last-minute dotting of i's and crossing t's.
- Allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.).
- By doing all of this work on a release branch, the develop branch is cleared to receive features for the next big release.

- Branch off a new release branch from develop
  - When develop (almost) reflects the desired state of the new release.
  - All features targeted for the release-to-be-built must be merged in to develop at this point in time.
- All features targeted at future releases should not merge in develop branch
  - They must wait until after the release branch is branched off.

RELEASE BRANCH CREATION

1. git checkout –b release-1.2 develop
2. ./bump-version.sh 1.2 //Increase the version to 1.2
3. git commit –a –m “Bumped version to 1.2”

Finishing a Release Branch

1. git checkout master
2. git merge –no-ff release-1.2
3. git tag –a 1.2

Deleting a Release Branch

1. git branch –d release-1.2

Finishing a Release Branch

1. git checkout develop
2. git merge –no-ff release-1.2

HOTFIX BRANCH

Always branches off from the **master**
Must merge back into **develop** and **master**
Branch naming convention **hotfix-***

- Hotfix branches are similar to release branches
  - Usually needed to prepare for a new production release, albeit unplanned.
- Branch off hotfix branch from corresponding tag on master when critical bug has to be fixed in production
- The essence is that work of team members (on the develop branch) can continue, while another person is preparing a quick production fix.

HOTFIX BRANCH

1. git checkout –b hotfix-1.2.1 master
2. ./bump-version.sh 1.2.1 //Increase the version to 1.2.1
3. git commit –a –m “Bumped version to 1.2.1”

Finishing a Hotfix Branch

1. git checkout master
2. git merge –no-ff hotfix-1.2.1
3. git tag –a 1.2.1

Deleting a Hotfix Branch

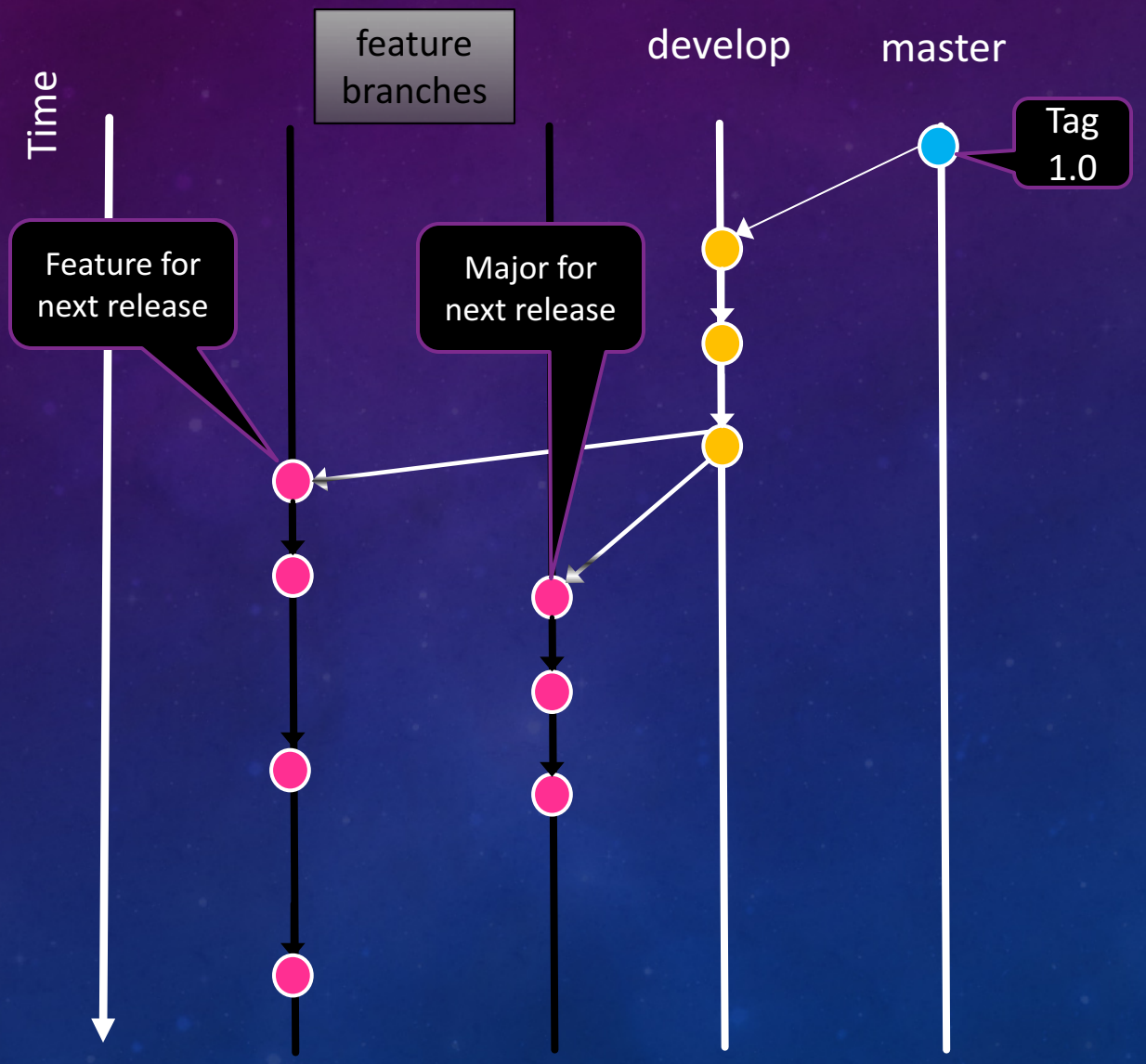
1. git branch –d hotfix-1.2.1

Finishing a Hotfix Branch

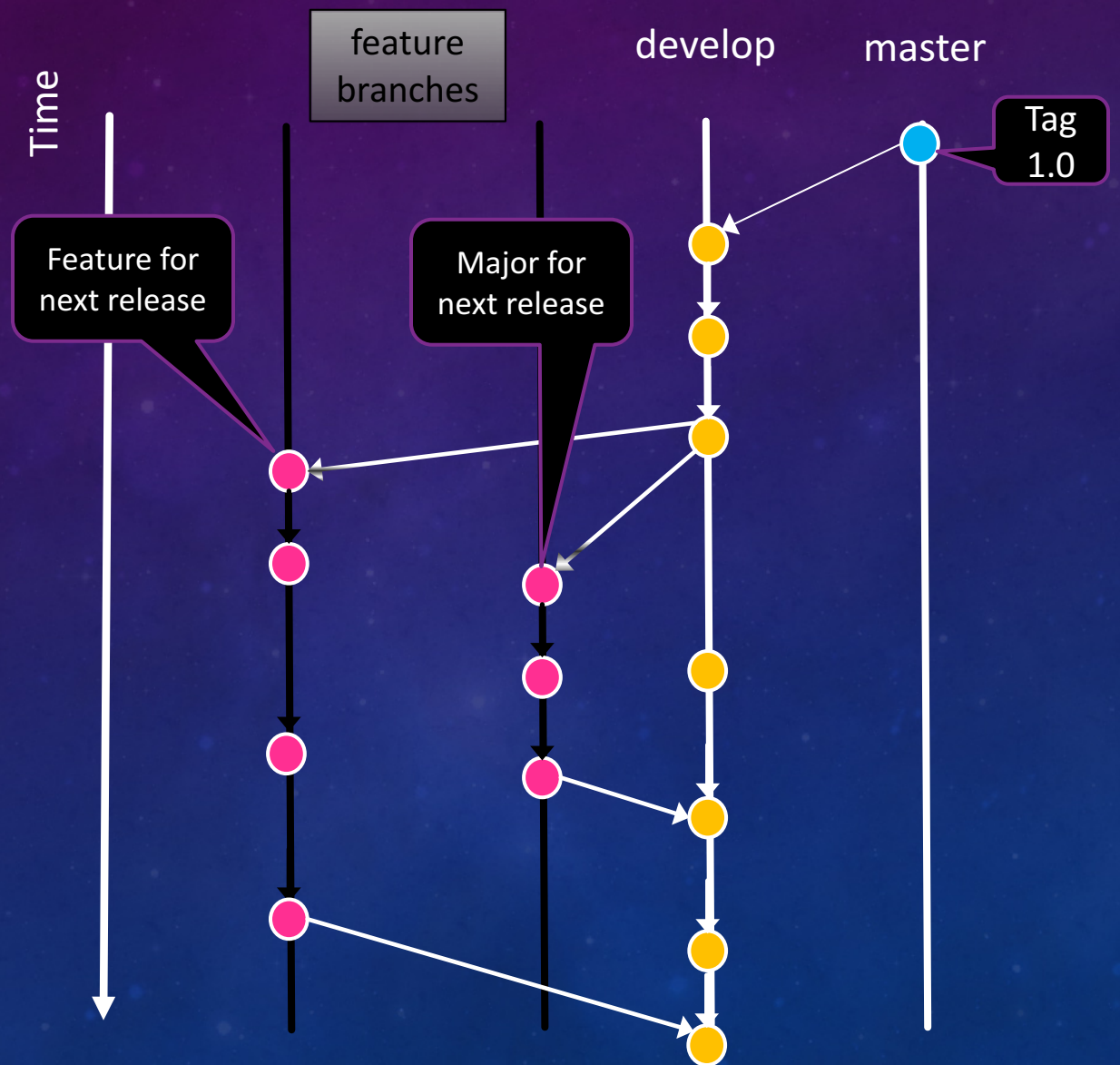
1. git checkout develop
2. git merge –no-ff hotfix-1.2.1

When a release branch currently exists, the hotfix changes need to be merged into that release branch, instead of develop

GIT FLOW: CREATING FEATURE BRANCH



MERGING FEATURE BRANCH TO DEVELOP



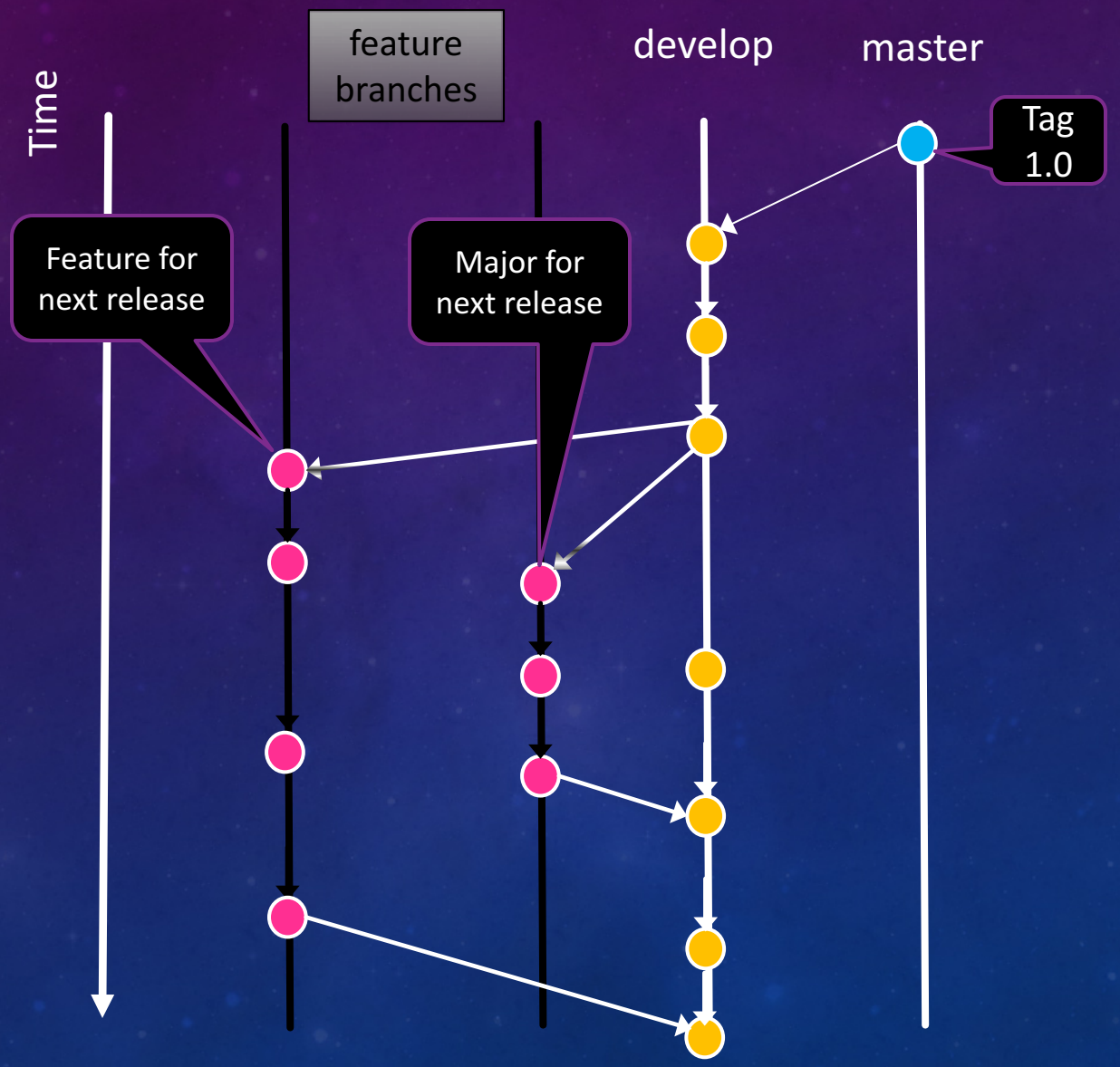
MERGING A FEATURE BRANCH INTO DEVELOPMENT

1. git checkout development
2. git checkout –b JIRA-105
3. work on the feature
4. commit your changes to the branch
5. git checkout development
6. git pull origin development
7. If no new changes are received in the pull
8. git merge -- no-ff JIRA-105
9. git push origin development
10. git branch –d JIRA-105

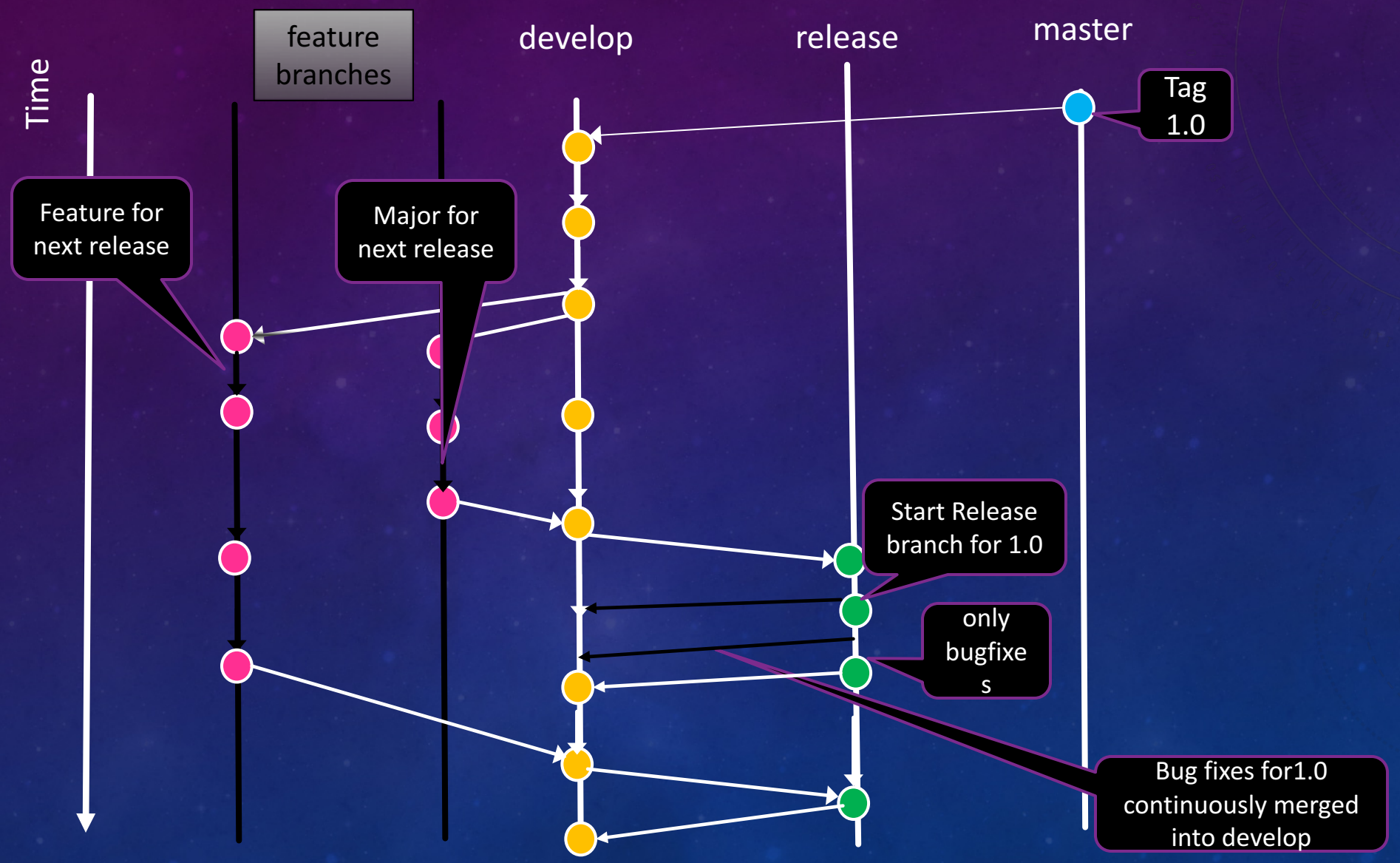
MERGING A FEATURE BRANCH INTO DEVELOPMENT

1. …
6. …
7. If some new changes are received in the pull
8. git checkout JIRA-105
9. git rebase development // Rebasing the current branch JIRA-105 with development
10. git checkout development
11. git merge -- no-ff JIRA-105
12. git push origin development
13. git branch –d JIRA-105

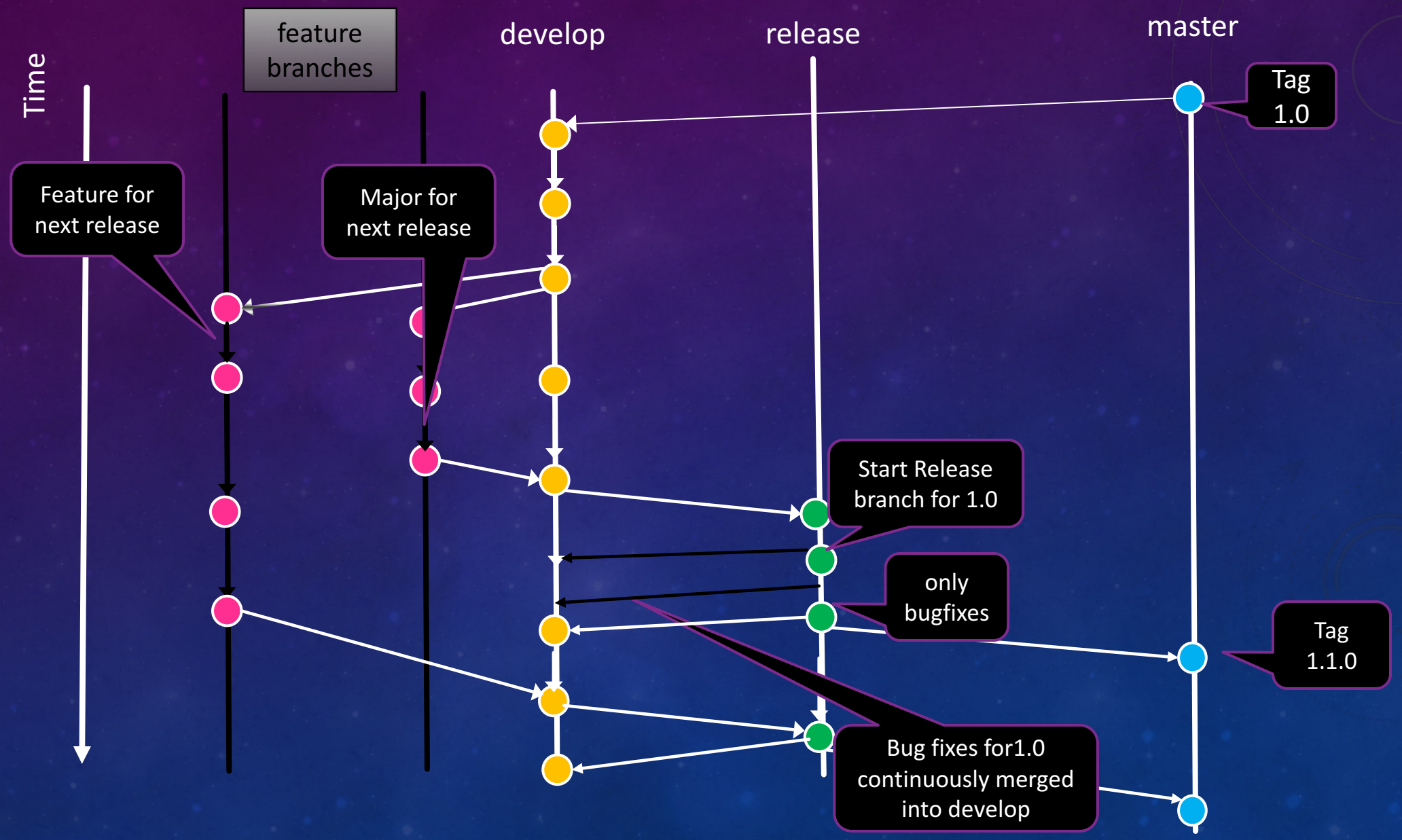
MERGING FEATURE BRANCH TO DEVELOP



MERGING FEATURE BRANCH TO DEVELOP



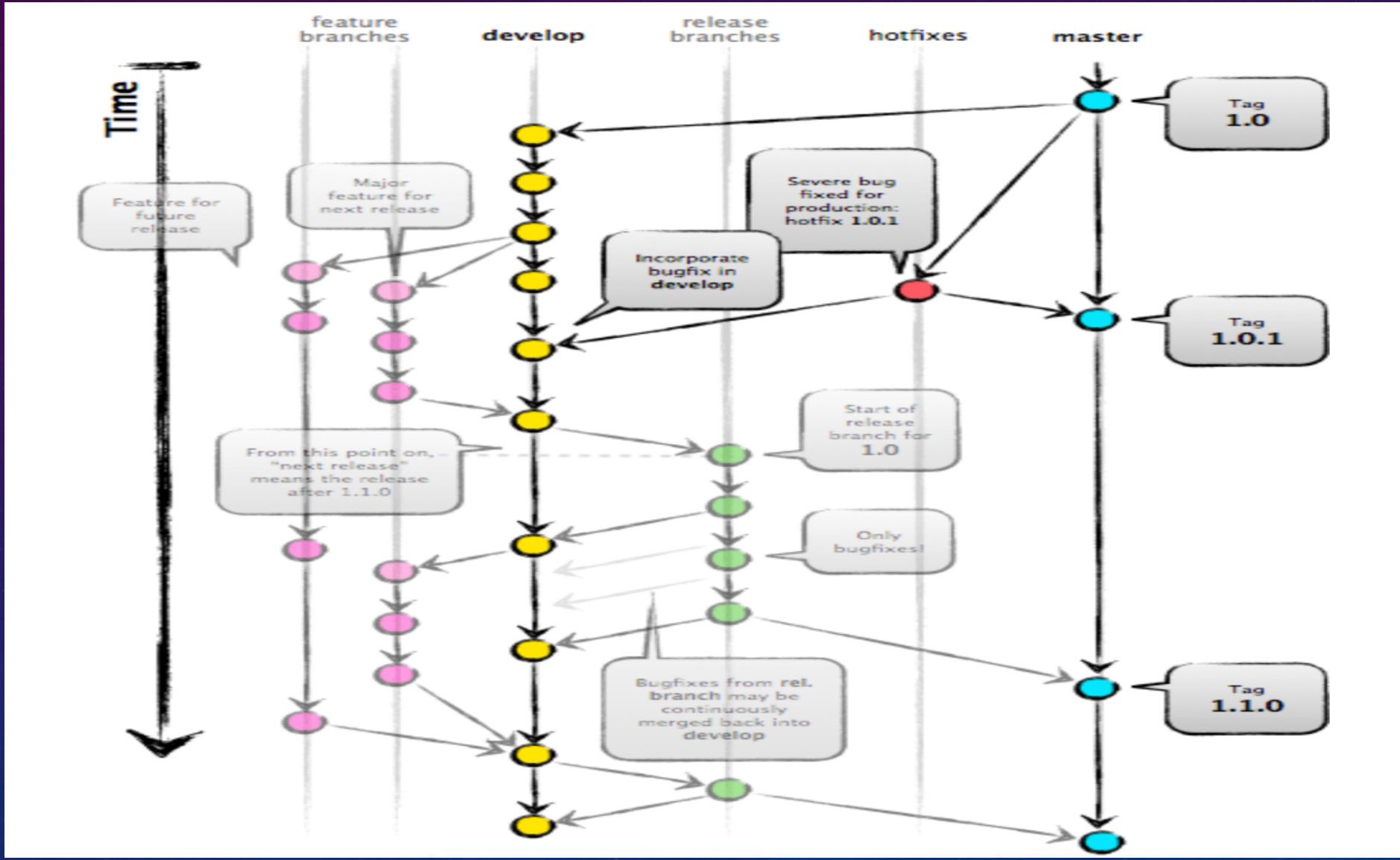
MERGING RELEASE BRANCH TO MASTER



RELEASING TO PRODUCTION

1. git checkout development
2. git checkout –b release-2.1
3. up the application version and commit this change
4. git push origin release-2.1
5. test the release branch on the staging environment
6. If there are any issues, fix them only on release branch
7. Once satisfied with release 2.1
8. git checkout master
9. git merge –no-ff release-2.1
10. git tag v2.1
11. git push origin master
12. git push –tags
13. Deploy the master branch to the production

CREATION AND MERGING OF HOTFIX BRANCH



ADDRESSING THE PRODUCTION ISSUES

Once a master has been deployed to production and there is some issue on production

1. git checkout master
2. git checkout –b hot-fix-2.1.1
3. increase the application version and commit this change
4. Fix the Issue and commit this change

- Merge the hotfix branch on master & development if there is no active release branch
- If there are any active release branch then merge it on master and the active release branch

ADDRESSING THE PRODUCTION ISSUES

Merging the hot fix on the master

1. git checkout master
2. git pull origin master
3. if there are no new changes:
4. git merge –no-ff hotfix-2.1.1
5. git push origin master
6. git branch –d hotfix-2.1.1

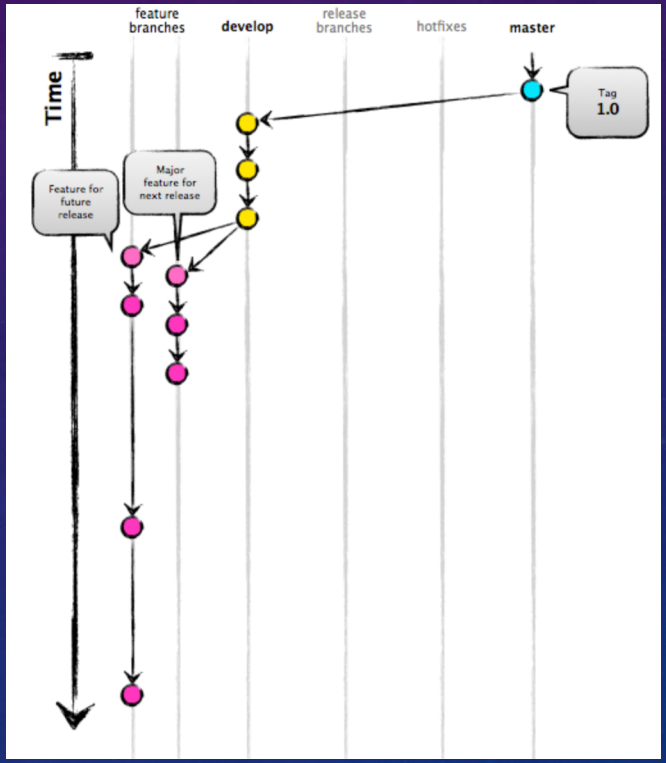
If there are changes in master then rebase the hotfix and then merge

ADDRESSING THE PRODUCTION ISSUES

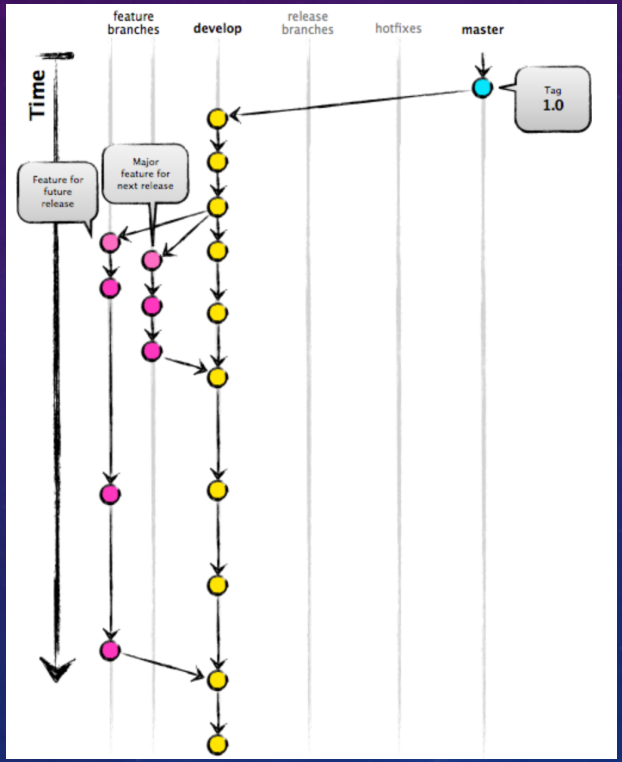
Merging the hot fix branch on development

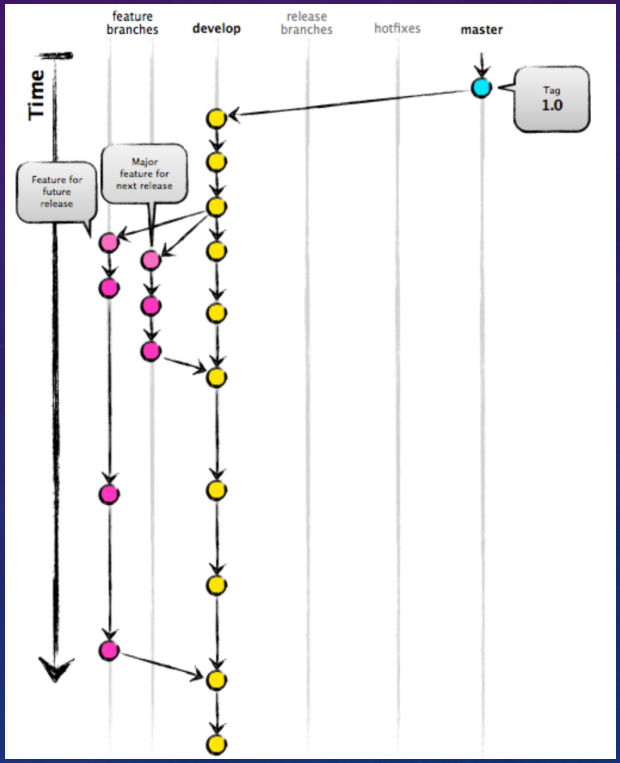
1. git checkout development
2. git pull origin development
3. if there are no new changes:
4. git merge –no-ff hotfix-2.1.1
5. git push origin development
6. git branch –d hotfix-2.1.1

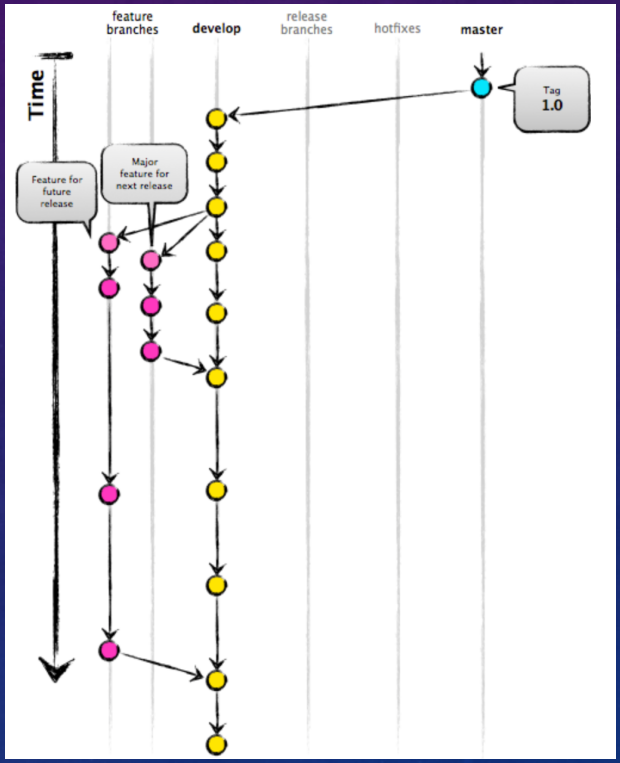
If there are changes in development then rebase the hotfix and then merge


feature branches
develop
release branches
hotfixes
master
Time
Tag
1.0
Feature for future release
Major feature for next release

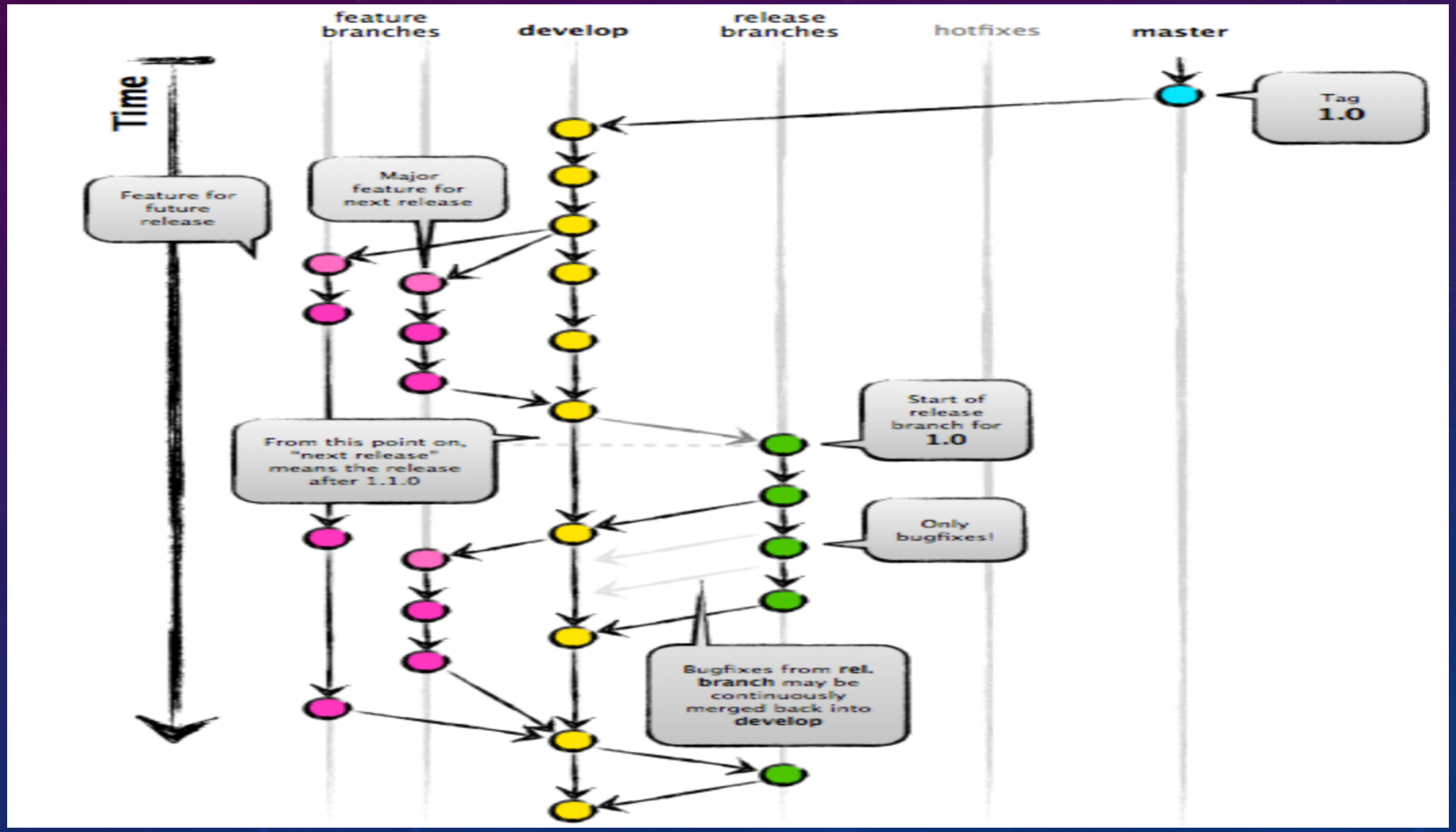
MERGING FEATURE BRANCH TO DEVELOP

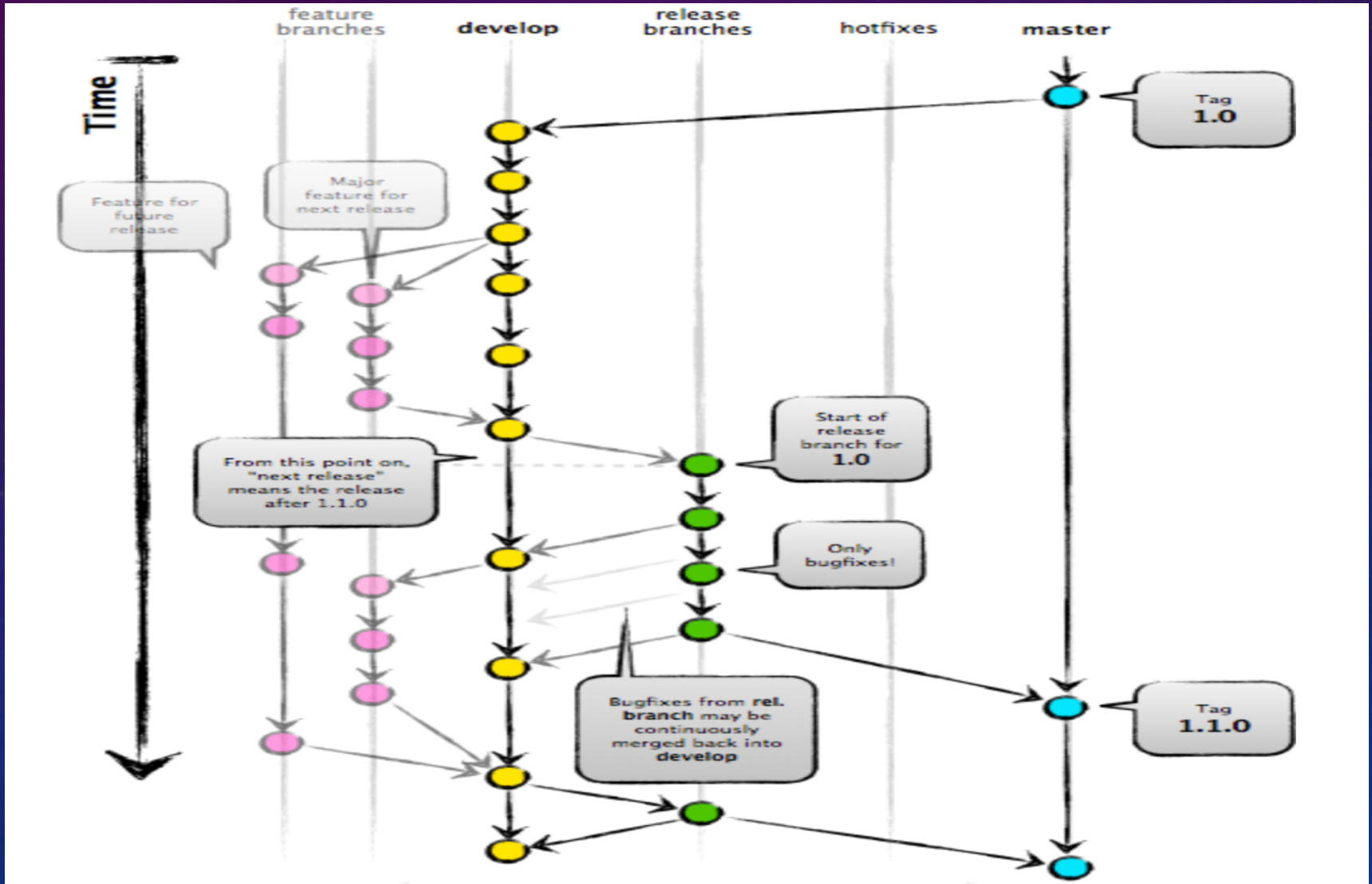



feature branches
develop
release branches
hotfixes
master
Time
Tag
1.0
Feature for future release
Major feature for next release


feature branches
develop
release branches
hotfixes
master
Time
Tag
1.0
Feature for future release
Major feature for next release

RELEASING FROM DEVELOP BRANCH




feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0

Can we still fix a bug for the upcoming Release?

How do we do Hotfix for the current version?

Is the code for that Feature complete?

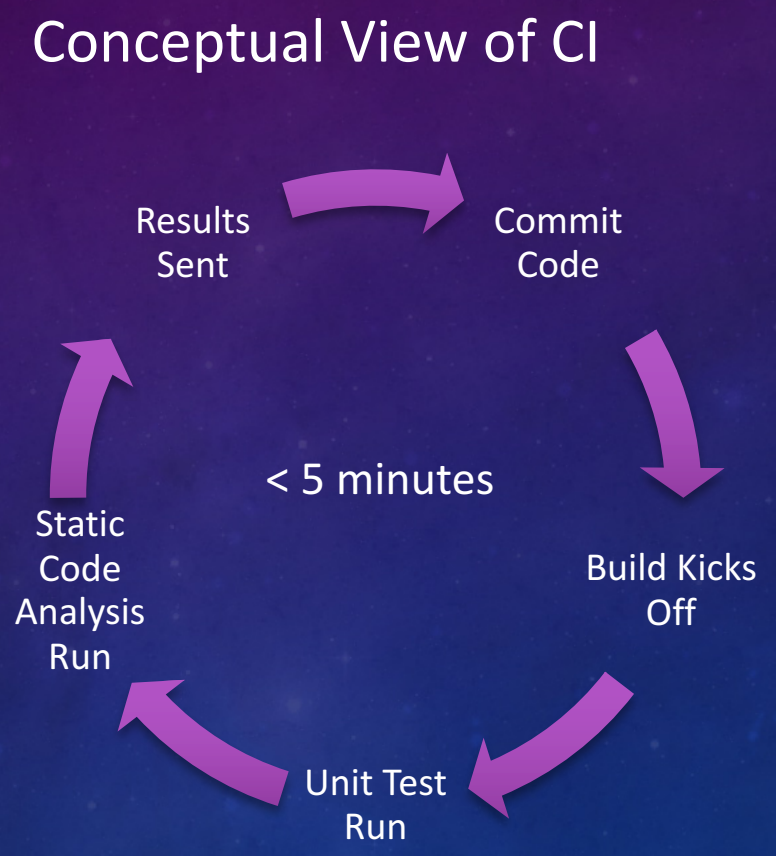
Has everyone Reviewed the code for this feature?

FEW RULES THAT ONE SHOULD FOLLOW

- Never pull on one branch from another. In order to get those changes, we do a rebase
- Delete the feature branches, release branches and hotfix branches once they are merged into the desired branches
- Never commit again on a branch which has already been merged into the destination branch
- Use feature branch names like ticket numbers, they help in quickly identifying the purpose of the branch

CI/ CD PRINCIPLES

WHAT IS CI?


Conceptual View of CI
Commit Code
Build Kicks Off
Unit Test Run
Static Code Analysis Run
Results Sent
< 5 minutes

CI DEFINITION

a fully automated and reproducible build, including testing, that runs many times a day

-- Martin Fowler

INSPECTION

- Code Coverage
- Code Complexity
- Copy’n’Pasting
- Architectural Standards
- Lint Checking
- Static Analysis

LIFE WITHOUT CI

- Unstable Code and Difficulty in Integration
- Many build errors reported
- Overdependence on few individual for building entire project
- Getting “Test Ready” product is quite hard
- Long Feedback cycle
- Same bug keeps coming back
- Productivity loss for entire team

LIFE WITH CI

- Simple and Repeatable Build Process
- Elimination of Human error
- Bugs stay fixed
- Rapid Feedback cycle
- Increase In Developers productivity

RISK REDUCTION

- What is in the build?
- What has changed between builds?
- How do I verify functionality?

REDUCE REPETITIVE PROCESS

- DRY
- Script everything
- Use build and configuration tool to achieve it

PRODUCE DEPLOYABLE SOFTWARE

- It’s what we are here to do
- Always in a “Demo Ready” state
- No last administrative tasks to get the product ready
  - Installer is built
  - Documentation is ready
  - Changelog is also captured

CHALLENGES WITH CI ADOPTION

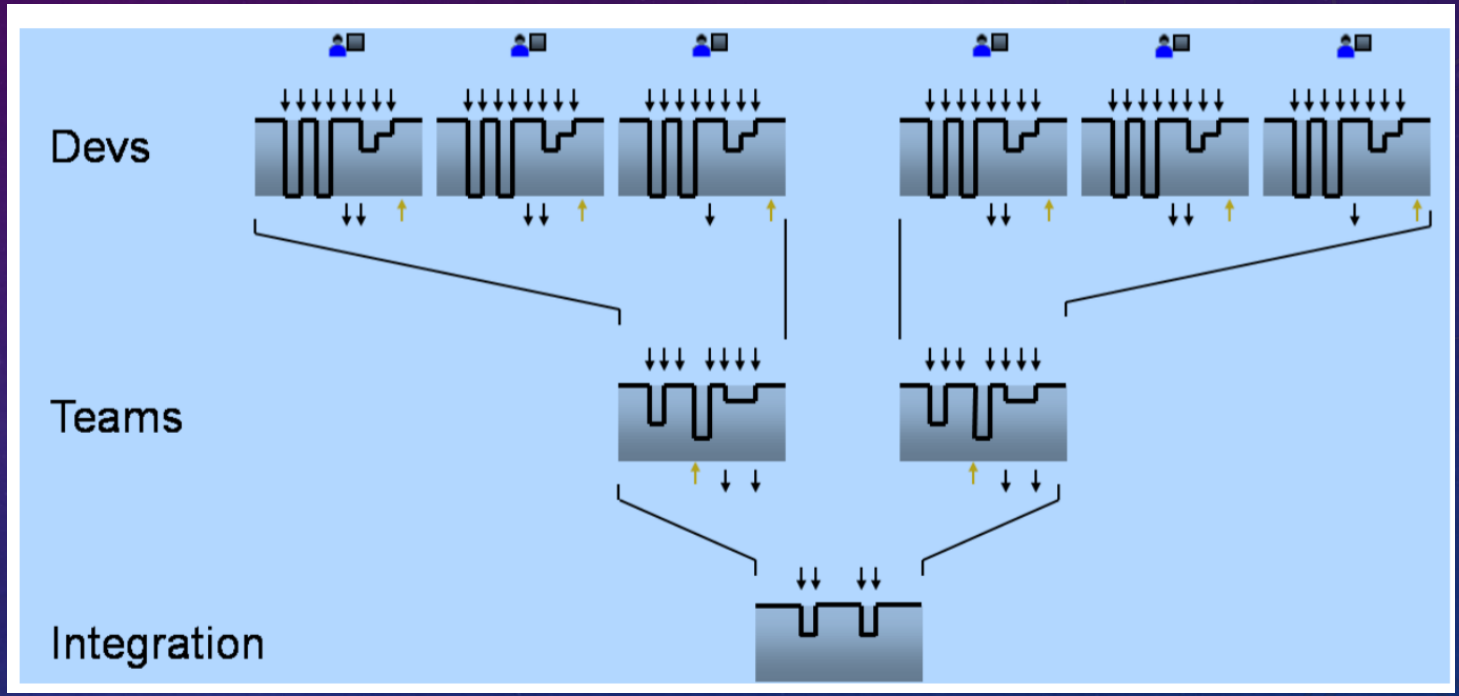
- Cultural Challenges
- Fear of Commitment
  - Developers and Testers need to commit changes frequently
- Fear of Blame
- Legacy System
- Structural Challenges
- Testability
  - Modularity?
  - Coupling to Underlying platform?
  - Requires external services to be up and available for testing?
- Tight Coupling
- CI Tool Stability Challenges

CI SYSTEM

- CI system must be as well maintained as the Source Control System
- CI system should perform builds very quickly
- No false alarms. False alarms can cause the team loosing confidence in CI system

MULTI STAGE CI

- Reduce the amount of Integrations Required
- Teams integrate as modules are marked as ready



CI TOOLS : WEB STACK

Build Tool

- Maven
- Gradle

Unit Testing

- Junit 4/5
- Mockito

Integration Testing

- Selenium

BDD

- Cucumber
- Jbehave
- Fitnesse

Code Review

- Crucible/ Stash

Source Repository

- Git
- Svn

Artifact Management

- Nexus/Artifactory

Quality Reporting

- Sonar (OSS/PSS)

CI Tool

- Teamcity
- Jenkins
- Others

Capacity Testing

- Gatling
- Jmeter
- Load Runner

Security Testing

- ZAPTest

CI TOOLS : DATABASE STACK

Build Tool

- Maven
- DdDeploy
- Flyway
- Liquibase

Unit Testing

- pl/sql unit
- utplsql
- DBFit

BDD

- Fitnesse/DBFit

Code Review

- Crucible/ Stash

Source Repository

- Git
- Svn

Quality Reporting

- Sonar (OSS/PSS)

CI Tool

- Teamcity
- Jenkins
- Others

Capacity Testing

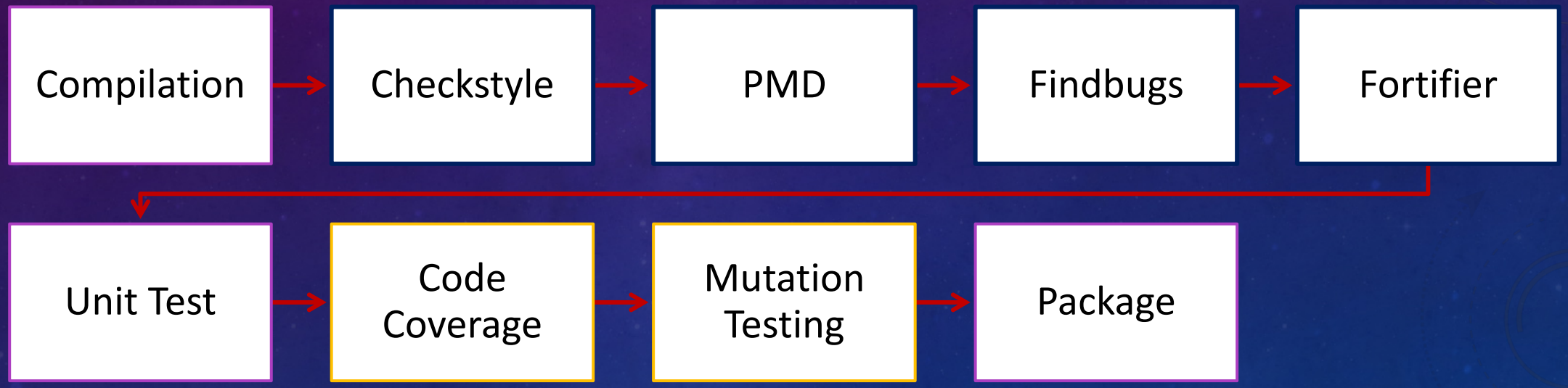
- Jmeter
- Load Runner

CI PRACTICES

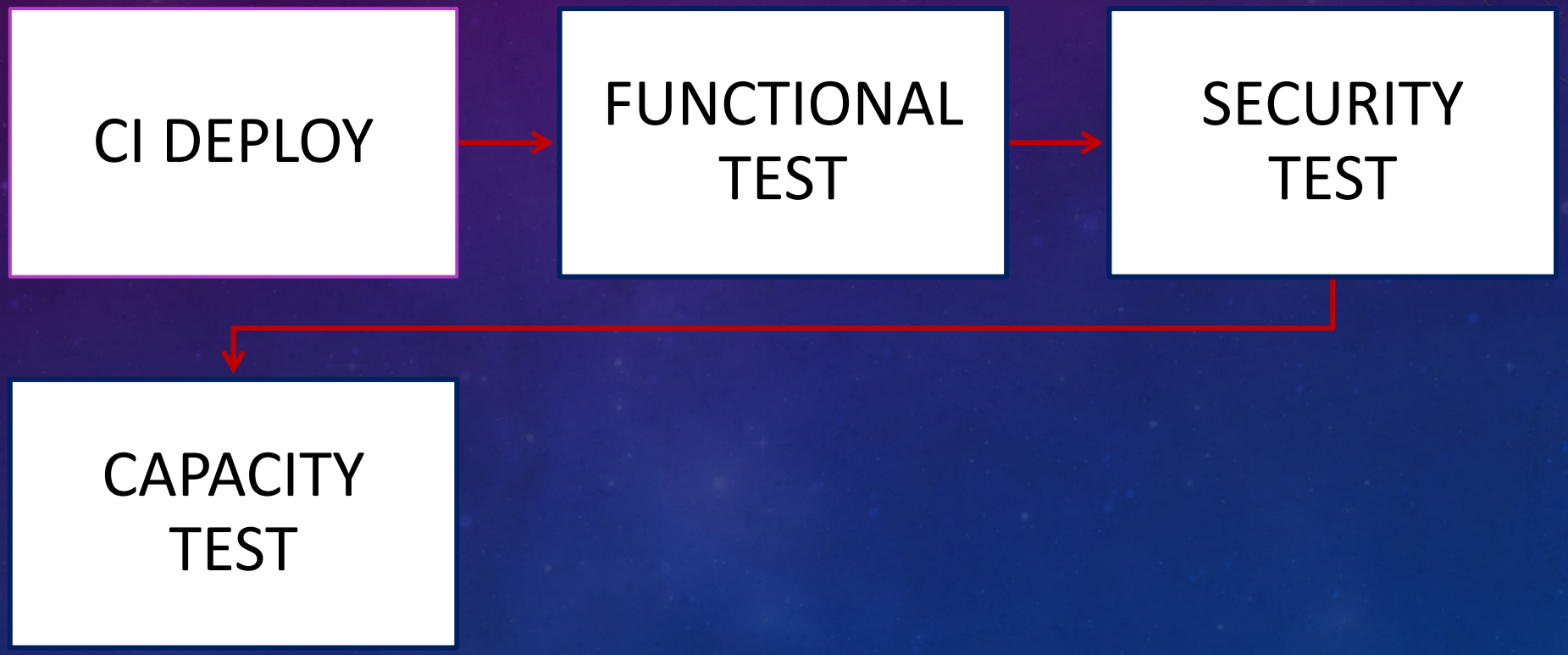
- Commit Frequently. Commit at each logical implementation
- Regularly sync your code with Source Repository
- Never commit to a failed build
- Own the responsibility of fixing a build quickly
- Single Source Repository
- Commit Often
- Make Your Build Self-Testing
- Automate the Build
- Build Fast

DEVOPS PIPELINE

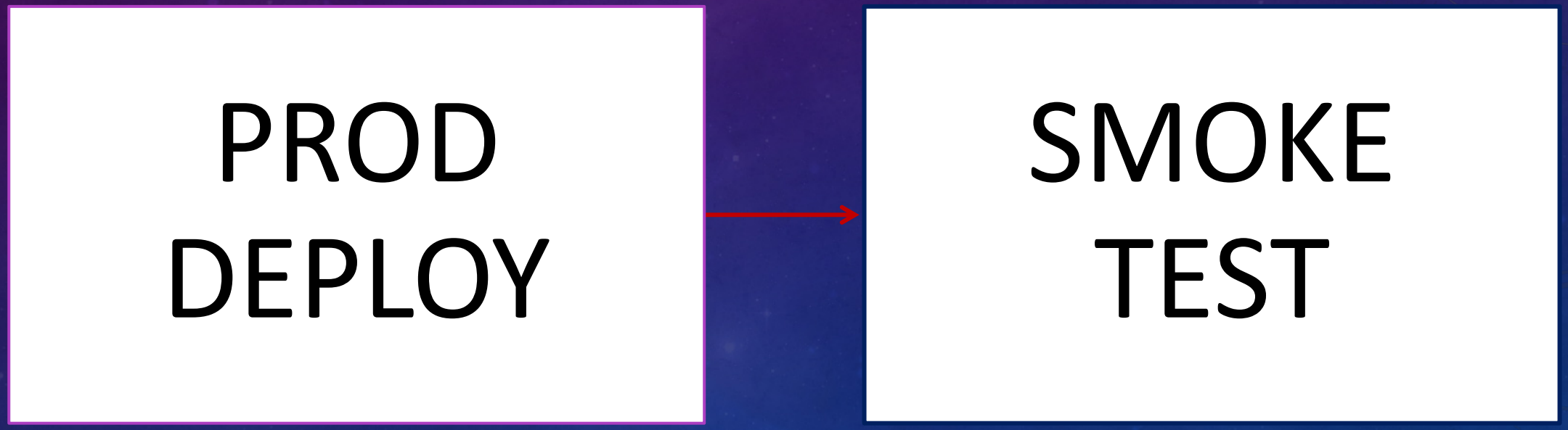
COMMIT STAGE PIPELINE: JAVA BASED PROJECT



ACCEPTANCE STAGE PIPELINE



DEPLOYMENT STAGE PIPELINE



DATABASE CI CHALLENGES

- Unlike other software components such as code or compiled code, a database is a collection of files
- Database is container of our valuable asset – business data, application content, customer transactions, etc.
- To promote database changes, a transition code needs to be developed
  - Scripts to handle database schema structure, database code (procedures, functions, etc.) and content used by the application (like metadata, lookup content or parameter tables)
- Database deployment is not only about applying the source code. It requires something more the just installing of source code

DATABASE DEPLOYMENT TOOL OPTION

Open Source

- Liquibase
- DbDeploy
- Flyway

Commercial

- RedGate
- DbMaestro
- MariaDB

MULTIPLE CHOICE

Option 1

1

The largest collection of cutout people characters!

100,000+ cutout people images with multiple poses, emotions, and props. Just drag and drop them into your project.

Option 2

2

Resizable characters images that can be edited!

Images that have been placed in your PowerPoint project can be edited, cropped, and resized.

Option 3

3

Illustrate and re-enforce project topics!

Characters can help people remember the information that you are sharing in your projects and presentations.

Option 4

4

Stand out from the crowd with cutout characters.

Boost your project to the next level with cutout characters. Each character comes with multiple poses, props, and emotions.

JENKINS

AGENDA

- Introduction
- Installation
- Plugins
- Build Job Configuration
  - Freestyle
  - Pipeline
- Jenkins Master/Slave Installation
- Build Pipeline Configuration
- Github Integration

JENKINS

- Open Source CI Server
- Easy to Install and Use
- Active Open Source Community
- Several Plugins to support all sorts of requirements

JENKINS INSTALLATION

- Go to download page https://jenkins.io/download/
- Simplest installation is to download the war file
- java -jar jenkins.war
- Go to http://localhost:8080

- See the other mode of installation when we setup master/slave nodes on aws

PLUGINS

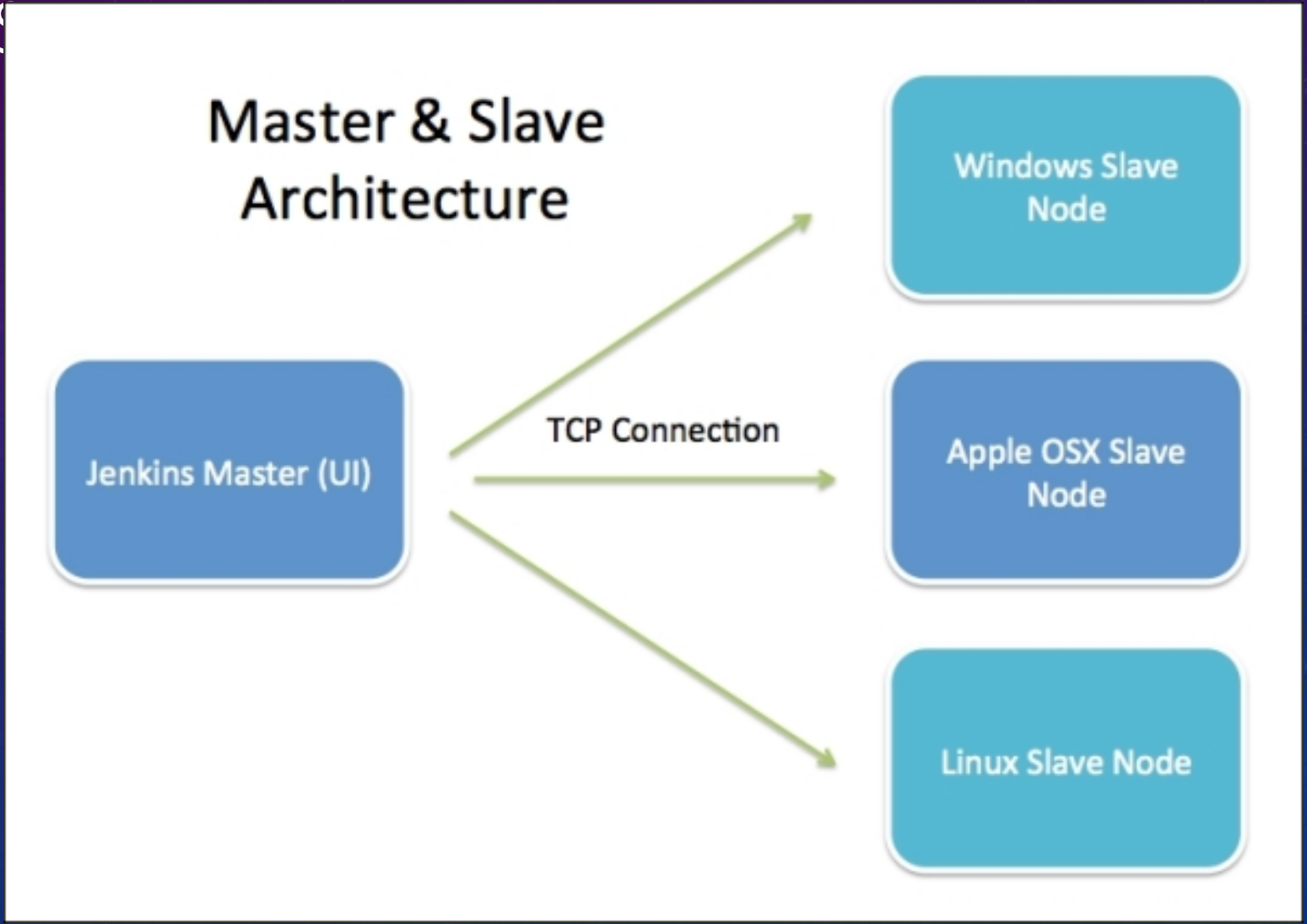
- Go to http://localhost:8080/pluginManager

- See the usage of some common plugins later on

DUMMY JOB CREATION

NEED FOR MASTER/S

- Limitation of running all jobs on Master nodes
- Master/Slave Architecture Model

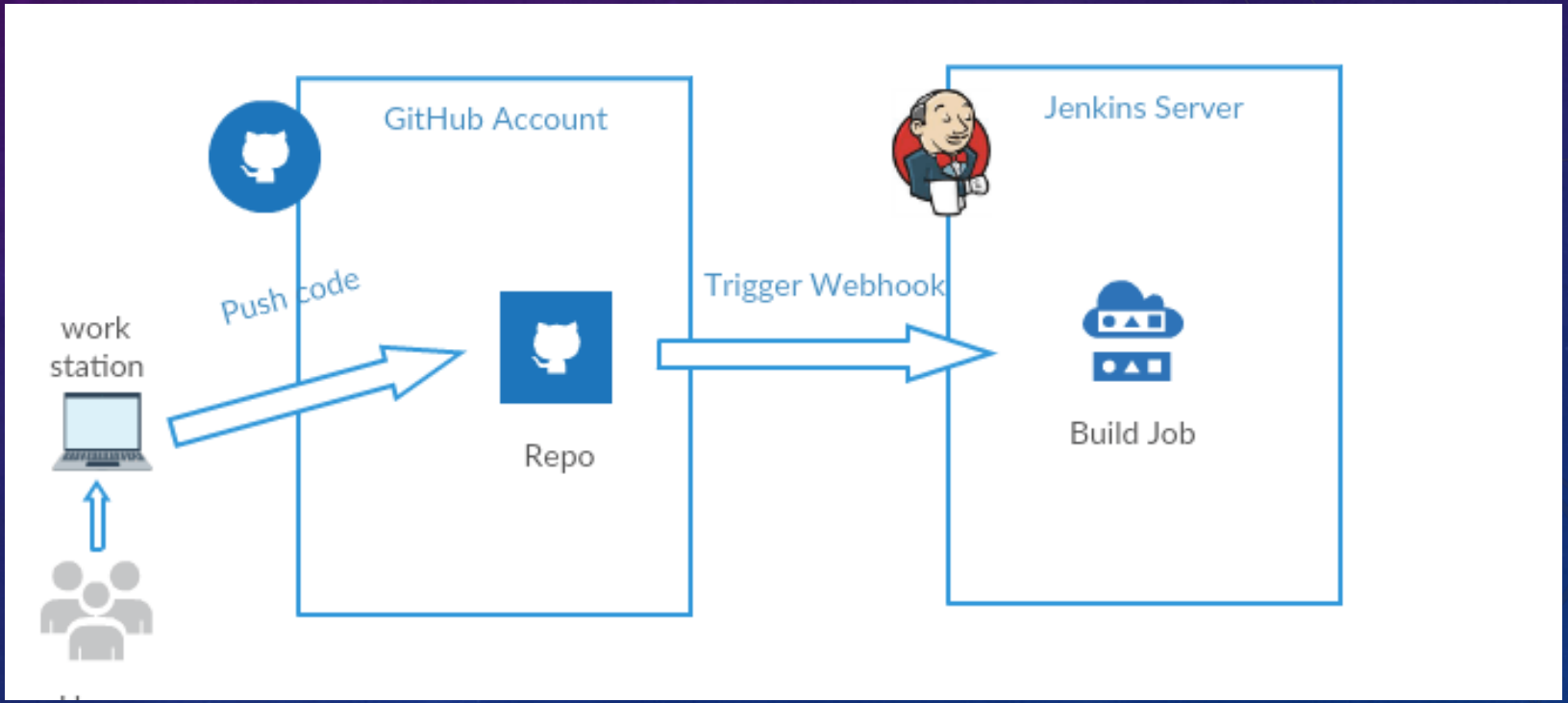


CONFIGURING GLOBAL TOOLS/SYSTEM

- JAVA
- Maven
- SonarQube

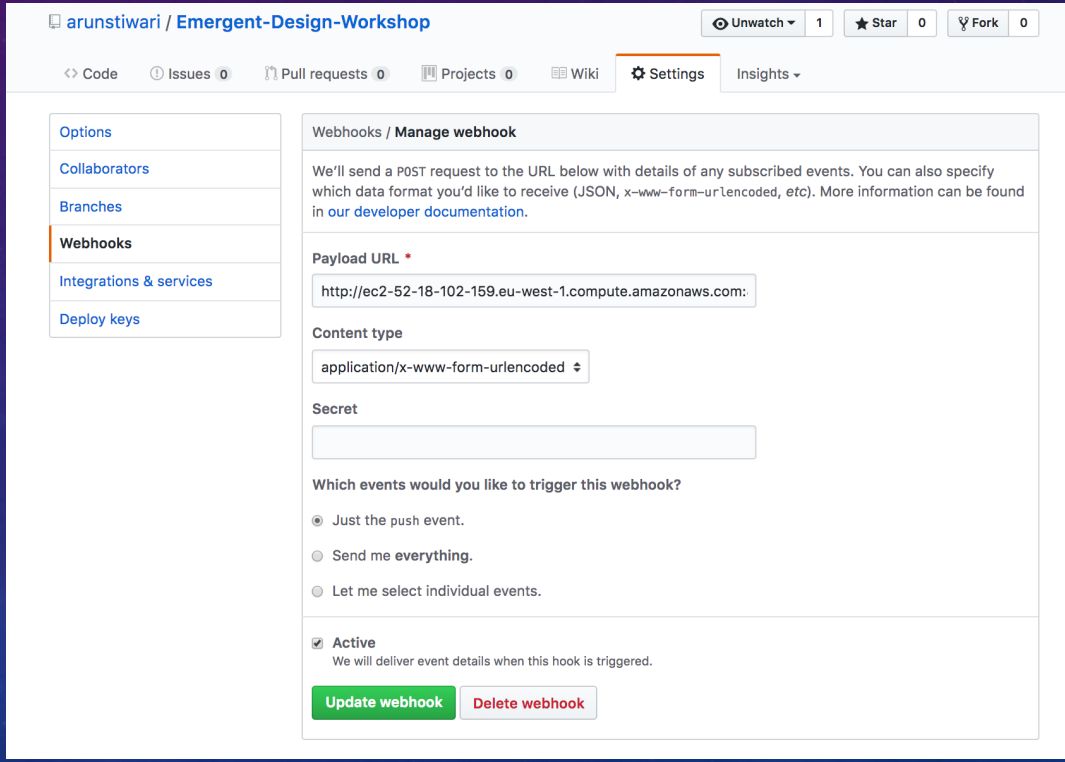
CONFIGURING GITHUB WEBHOOK

CONFIGURING GITHUB WEBHOOK



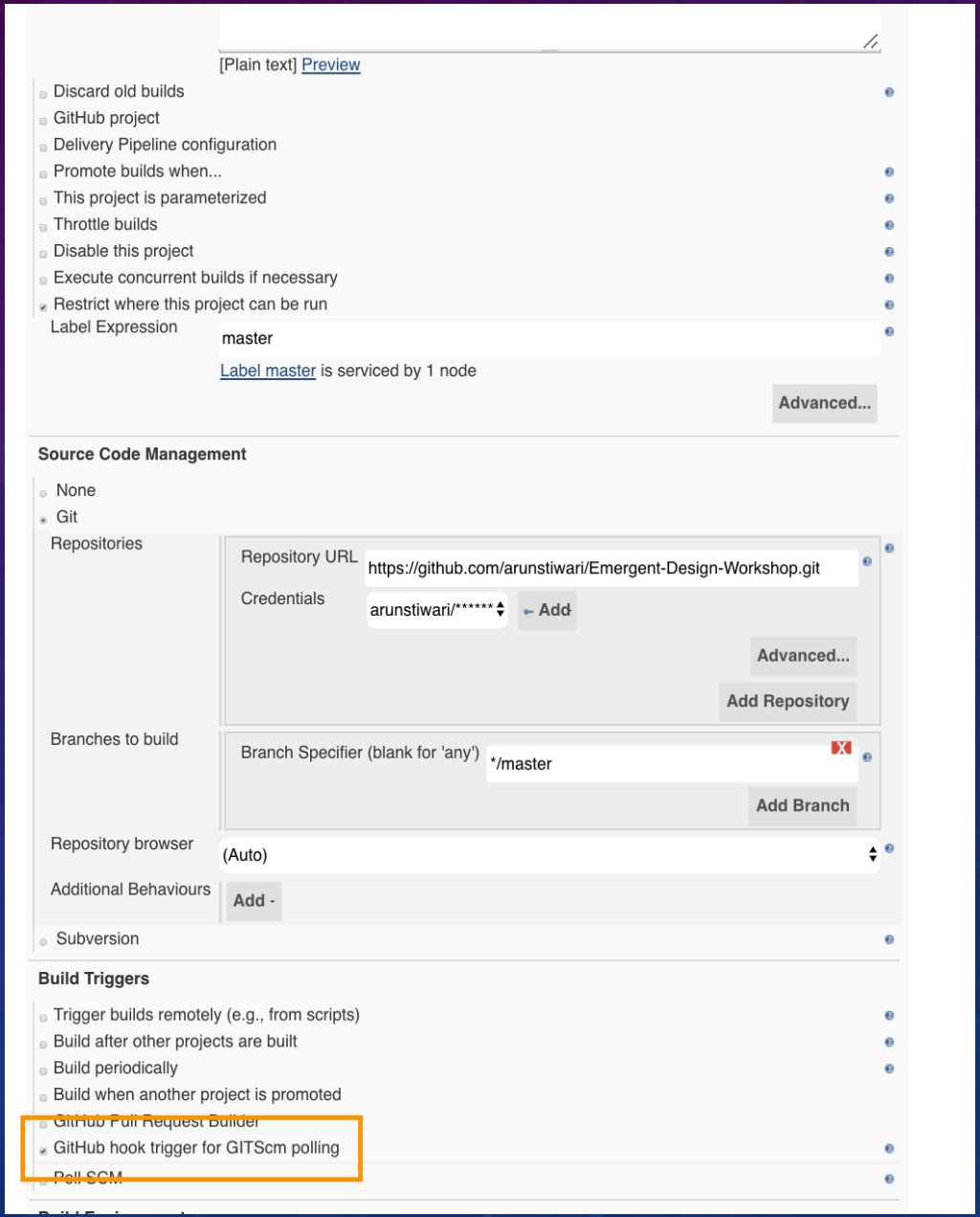
WEBHOOK CONFIGURATION

- Go to Github repo Settings section
- Add Payload URL (Jenkins server url)



TEST TRIGGER JOB

- Configure Job to test Github hook



CONFIGURING PIPELINE VIEW

PIPELINE VIEW

- More Visual Representation of Dependent Job View

CREATE A PIPELINE VIEW

Name: My-Pipeline

Description: CI Pipeline

[Plain text] Preview

Filter build queue

Filter build executors

Build Pipeline View Title: CI Pipeline

Pipeline Flow

Layout: Based on upstream/downstream relationship

This layout mode derives the pipeline structure based on the upstream/downstream trigger relationship between jobs. This is the only out-of-the-box supported layout mode, but is open for extension.

Upstream / downstream config

Select Initial Job: Job1

Trigger Options

Build Cards: Standard build card

Use the default build cards

Restrict triggers to most recent successful builds: Yes No

Always allow manual trigger on pipeline steps: Yes No

Display Options

No Of Displayed Builds: 2

Row Headers: Just the pipeline number

Show just the build pipeline number

Column Headers: All build variables and parameters

Show all variables in the current build, and obfuscate sensitive values. Variables are drawn from `Build#getBuildVariables()`.

Refresh frequency (in seconds): 3

URL for custom CSS files

Console Output Link Style: Lightbox

OK Apply

CONFIGURING PIPELINE JOB

PIPELINE JOB

Jenkins Pipeline (or simply "Pipeline" with a capital "P") is a suite of plugins which supports implementing and integrating *continuous delivery pipelines* into Jenkins.

- Uses DSL syntax
- Versioned Control as Jenkinsfile
- Jenkinsfile syntax needs to be learnt
- More can be learnt at https://jenkins.io/doc/book/pipeline/

Jenkinsfile (Declarative Pipeline)

```
pipeline {
   agent any
  stages {
    stage('Build') {
      steps { sh 'make' }
    }
    stage('Test'){
      steps { sh 'make check' junit 'reports/**/*.xml' }
   }
   stage('Deploy') {
     steps { sh 'make publish' }
  }
 }
}
```

DEMO PIPELINE

- Covers Unit Test
- Coverage Using JaCoCo
- Checkstyle
- SonarQube
- Reports Generation

RECAP

QAI

SONARQUBE INSTALLATION

- https://www.sonarqube.org/downloads/
- https://www.sonarsource.com/products/codeanalyzers/sonarpython.html
- https://www.sonarsource.com/products/codeanalyzers/sonarcsharp.html
- sudo apt install openjdk-8-jdk

- Go to the **Administration** tab -> **Marketplace** -> **Installed**
- Confirm that SonarPython plug-in is installed, if not install it.
- Restart the SonarQube server if needed.

CONFIGURING SONAR SCANNER ON JENKINS

1. We will need to install *Sonar plug-in* for Jenkins
   - Open your Jenkins CI server on your browser and login as administrator.
   - Go to: **Manage Jenkins** -> **Manage Plugins** -> **Available**
   - Search and Install *'Sonar'*, if you cannot find it
   - Download latest Sonar.hpi plugin
   - Select **Advanced** -> **Upload Plugin** -> **Restart Jenkins**
2. Configure your **SonarQube Servers** on JenkinsGo to **Manage Jenkins** -> **Configure System**
   - Scroll down to the **SonarQube servers** section
   - Enable: *Enable injection of SonarQube server configuration as build environment variables*
   - click on **Add SonarQube**, and add the values you're prompted for
   - Name: SonarQube
   - Server Authentication token obtained, elif you do not have the token read User Guide - User Token

ENABLE ANALYSIS WITH SONARQUBE SCANNER

- In order to trigger SonarQube analyses with the SonarQube Scanner, we will need to define our sonarqube scanner instance on Jenkins global configuration.
  - Open your Jenkins CI server and login as administrator
  - Go to: **Manage Jenkins** -> **Global Tool Configuration**
  - Scroll down to the **SonarQube Scanner** configuration section
  - Click on **Add SonarQube Scanner** and
  - Enable: **Install Automatically** or choose to point to an already installed version of SonarQube Scanner (uncheck '*Install automatically*') or tell Jenkins to grab the installer from a remote location (check '*Install automatically*')
  - Save and exit
-

List of SonarScanner for MSBuild installations on this system

SonarQube Scanner

SonarQube Scanner installations

SonarQube Scanner

Name: SonarQube Scanner

☑ Install automatically

Install from Maven Central

Version: SonarQube Scanner 3.2.0.1227

Delete Installer

Extract *.zip/*.tar.gz

Label: SonarQube

Download URL for binary archive: https://sonarsource.bintray.com/Distribution/sonar-scanner-cli/sonar-scanner-cli-3.2.0.1227-linux.zip

Subdirectory of extracted archive:

Delete Installer

Add Installer

CONFIGURE JENKINS BUILD

- Once we have configured SonarQube, we can now run a SonarQube build
  - Go to your Jenkins build,
  - **Configure** -> **Build Environment** -> Enable: **Prepare SonarQube Scanner environment**
  - -> **Build** -> **Add build step** -> **Execute SonarQube Scanner**
- Now we need to either point the executer to a path with project properties or add our own Analysis properties.
  - Create *sonarscanner properties*
  - Let *JDK: inherit from Job*